ESD-TR-71-345

QUEUEING NETWORK MODELS OF MULTIPROGRAMMING

Jeffrey P. Buzen

August 1971

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

(Prepared under Contract No. F19628-70-C-0217 by Harvard University, Cambridge, Massachusetts 02138.)

LEGAL NOTICE

When U. S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy

ESD-TR-71-345

QUEUEING NETWORK MODELS OF MULTIPROGRAMMING

Jeffrey P. Buzen

August 1971

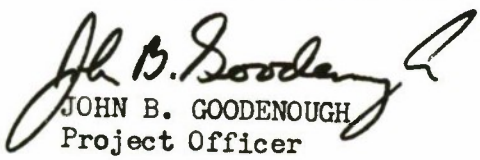
DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

(Prepared under Contract No. F19628-70-C-0217 by Harvard University, Cambridge, Massachusetts 02138.)

FOREWORD

This report was prepared in support of Project 2801, Task 280102 by Harvard University, Cambridge, Massachusetts under Contract F19628-70-C-0217, monitored by Dr. John B. Goodenough, ESD/MCDT-1, and was submitted August 1971.

This technical report has been reviewed and is approved.



JOHN B. GOODENOUGH
Project Officer

ABSTRACT

A model is developed which represents the behavior of multiprogrammed computer systems in terms of a network of interdependent queues. This model, which is known as the central server model of multiprogramming, is first analyzed mathematically and then applied to three problems in operating system design. These are: the optimal choice of buffer size for tape-like devices; the optimal allocation of processing requests among a set of functionally equivalent peripheral processors such as disks and drums; the optimal selection of the degree of multiprogramming in demand paging systems.

A series of computational algorithms are developed to supplement the analytic work. These algorithms can be used to obtain the marginal distributions and expected queue lengths for a large class of queueing network models.

PREFACE

I am deeply indebted to my advisor Dr. Ugo O. Gagliardi for introducing me to the subject of queueing theory and for encouraging me to work in this area. In addition, Dr. Gagliardi's clear vision of the underlying nature of scientific research has provided a sharp focus of expectation throughout the course of this thesis.

I would also like to express my gratitude to the other members of my committee, Professors T.E. Cheatham, Jr., A.G. Oettinger and W.A. Woods, for the time they have invested in reading this thesis and for their pertinent comments. R.M. Kierr's extremely careful and thoughtful reading of an early draft of Chapter 4 is also appreciated. Finally, I would like to point out that a number of ideas in Chapter 7 were generously contributed by C.G. Moore and S.R. Kimbleton of the University of Michigan and by F. Baskett of the University of Texas. These contributions are explicitly acknowledged in footnotes and in the body of the text.

The work leading to this thesis was supported in part by the National Science Foundation through the Traineeship program. Support for the thesis itself was provided by the Veterans Administration under Chapter 34, Title 38 of the G.I. Bill and by the Electronic Systems Division, L.G. Hanscom Field, Bedford, Massachusetts under Contract No. F-19628-70-C-0217.

My wife Judy deserves special recognition for her unique sensitivity and consistent support during the preparation of this thesis. Her many intangible yet highly significant contributions are sincerely appreciated.

JEFFREY BUZEN

Watertown, Massachusetts
May, 1971

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SYNOPSIS

The major portion of this thesis is devoted to the development, analysis and application of the central server model of multiprogramming. This model represents the overall behavior of large scale multiprogramming systems in terms of a network of queues. Each processing element and active program in the system being modeled is explicitly represented. In addition, the effect on overall system performance of random variability in individual program behavior is implicitly taken into account.

The mathematical treatment of the central server model begins with a derivation of the steady state distribution. The properties of this distribution are then examined in a series of informal theorems and corollaries. Following this a number of highly efficient computational algorithms are developed for numerically evaluating the steady state distribution in specific instances. These algorithms, which are applicable to a wide class of queueing networks, make it possible to easily carry out computations which would otherwise be near or in some cases even beyond the limits of current technology. The value of these algorithms thus extends well beyond the context of the thesis itself.

In addition to these analytic and computational results, the central server model is also applied to three specific problems in computer systems analysis. These problems

involve the optimal specification of buffer size for tape-like devices, the optimal allocation of processing requests among a set of functionally equivalent peripheral processors such as disks and drums, and the optimal allocation of main memory in systems employing demand paging.

All three problems generated unanticipated results. In the first case it was discovered that, with the initial overhead per transfer held constant, optimal buffer size decreases as the transfer rate of the associated peripheral processor increases. Analysis of the second problem revealed that optimal performance is attained when the fastest processor is receiving more than its proportional share of processing requests and is in effect creating a system bottleneck. Finally it was shown in the third problem that in certain cases it is more important to have efficient page replacement algorithms in systems with fast page transfer processors than it is in systems with slow page transfer processors.

The contents of each chapter of this thesis may be briefly summarized as follows:

Chapter 1 presents a discussion of the merits of queueing network models and a guide to the remainder of the thesis.

Chapter 2 introduces some basic queueing theoretic notions and then reviews a total of fifty-five papers concerned with the application of queueing theory to computer systems analysis.

Chapter 3 briefly traces the development of analytic methods and models in the field of queueing network research.

Chapter 4 provides the basic motivation for the central server model and also examines some of the model's elementary properties.

Chapter 5 presents a derivation of the steady state distribution for the central server model and an examination of the analytic and computational aspects of this distribution.

Chapter 6 explores the applications of the central server model to problems of buffer size determination, peripheral processor utilization and page traffic balancing.

Chapter 7 develops a number of extensions to the basic central server model which make it possible to represent more general classes of systems.

Chapter 8 examines the relationship between this thesis and previous research. In addition a number of problems are presented for future consideration.

Appendix A discusses the nature of the exponential distribution with emphasis on the so-called 'memoryless' property.

Appendix B provides a detailed explanation of the powerful but little known solution technique which was used in Chapter 5 to obtain the steady state distribution for the central server model.

CHAPTER 1: INTRODUCTION

THE NEED FOR QUEUEING NETWORK MODELS

Large scale multiprogramming systems are typically composed of a number of individual processing elements such as computational processors, device controllers, data channels and so forth. These processing elements normally operate in parallel with one another subject to constraints generated by the programs which run on the system. That is, even though the processing elements which make up a multiprogramming system may be capable of fully parallel operation, the degree of parallelism which the system actually attains is always limited by the sequential nature of the processing requests that individual programs generate. Thus any model of a multiprogramming system must incorporate both parallel processing capabilities and sequential processing constraints.

Random variability also has a significant effect on the performance of multiprogramming systems. Essentially, this factor creates the possibility of queueing delays even though the average interval between arrivals at a system processor may be greater than the average service time per processing request. Queueing delays created by random variability work in conjunction with sequential processing constraints to further reduce the degree of parallelism in multiprogramming systems. The effect of this factor may be quite significant.

For example, in the case in which all active programs have requests pending for the same processing element at the same time, parallelism may entirely disappear. Hence any realistic multiprogramming model must include random variability along with parallel processing capabilities and sequential processing constraints.

One of the primary purposes of this thesis is to demonstrate that all three of these factors can be represented quite naturally within the framework of a queueing network model. In such a model each server in the network corresponds to an individual processing element, the path that a customer follows while moving through the network corresponds to the sequence of processing requests generated by a particular program, and the random variability in service times and customer movement corresponds to the random variability in the actual system. In addition the number of customers in the network at any time clearly corresponds to the degree of multiprogramming of the system being represented.

Queueing network models may be addressed to a number of problems in computer systems analysis. For example, it is possible to study the effects of various modifications in system hardware by utilizing the correspondence between actual processor speed and network service time or the correspondence between main memory size and number of customers in the network. In a somewhat different context, the correspondence between program behavior and the paths that cus-

tomers follow as they move through the network can be used to study problems such as the optimization of program structure with respect to system hardware. Finally, it is possible to study more complex problems such as the optimal allocation of main memory in systems with demand paging. Problems of this type involve the interaction of several system components and cannot be adequately treated by simpler models which take only one processing element into account.

Despite the obvious advantages of queueing network models, very few analyses of such models have appeared in the literature. This is no doubt related to the mathematical difficulties associated with the general analytic treatment of models of this type. However, in many specific cases of interest - including those considered in this thesis - it is possible to significantly reduce the mathematical complexity of the problem by applying a powerful solution technique which was originally developed by Jackson (48) in 1963 and then independently discovered by Gordon and Newell (41) shortly thereafter. Since this solution technique is not widely known within the field of computer systems analysis, its most significant aspects have been reproduced in Appendix B. It is hoped that the increased availability of this technique together with the examples and supplementary numerical algorithms developed in this thesis will generate additional interest in this area and will ultimately lead to a series of highly useful and revealing queueing network models.

ORGANIZATIONAL REMARKS

A brief summary of the contents of each chapter of this thesis is provided in the Synopsis. It should be apparent from this summary that Chapters 4, 5 and 6 present the bulk of the new material in the thesis. These chapters are entirely self-contained and should be readily understandable to anyone familiar with queueing theory and operating system fundamentals.

Readers more interested in practical applications may wish to restrict their attention to the section of Chapter 4 which deals with specification of the model, the section of Chapter 5 which deals with system performance, and the three examples in Chapter 6. The extensions discussed in Chapter 7 and the suggestions for further research presented in Chapter 8 should also be of interest to this group.

The more mathematically inclined readers will probably wish to read all of Chapters 4 and 5. However, the only application of real mathematical interest in Chapter 6 is the one dealing with peripheral processor utilization. In addition, any mathematically inclined reader not already familiar with the work of Jackson (48) and Gordon and Newell (41) should find Appendix B extremely valuable. The work of Jackson and Gordon and Newell is also discussed in more qualitative terms in Chapter 3.

The survey presented in Chapter 2 is self-contained and should provide a helpful introduction to students and other individuals entering this field of research. In addition, Chapters 3 and 8 contain more specialized surveys. All three of these chapters contain discussions of unsolved and potentially significant research problems.

CHAPTER 2: SURVEY OF THE APPLICATIONS OF QUEUEING THEORY TO COMPUTERS

ESSENTIALS OF QUEUEING THEORY

Queueing theory may be thought of as a collection of analytic techniques and mathematical results all related to the analysis of a particular abstract process. Essentially this process is one in which customers arrive at some service facility, present that facility with requests for service, and then leave the facility after their individual requests have been satisfied. In this general setting queueing theory deals with such questions as the number of customers at the facility at any time, the total amount of time required to process individual customers through the facility, and the nature of the periods during which the facility is continuously busy serving customers.

Random variability is one of the essential distinguishing features of all queueing systems. Basically, there are two ways such variability can enter: either in the time intervals between the arrival of successive customers, or in the amount of service that individual customers request. In most queueing systems both these factors are assumed to be non-constant random variables. However, there are some cases of interest in which one of these factors is constant. Systems in which both factors are constant or cycle deterministically through a given set of values are not tradi-

tionally regarded as falling within the realm of queueing theory since a different set of mathematical techniques is required for their analysis.

In the standard terminology of queueing theory, the length of the intervals between the arrival of successive customers is determined by the arrival process and the amount of service that each customer requests is determined by the service time distribution. If the inter-arrival intervals are independent of each other and exponentially distributed (see Appendix A), the arrival process is known as a Poisson process. This process is of fundamental importance in queueing theory because of its mathematical simplicity and its reasonably close correspondence to many physical situations. If the service time distribution is also exponential, further simplifications are introduced, but it is not always necessary to make this additional assumption in order to obtain significant results.

A queueing system is characterized by specifying an arrival process, a service time distribution, and a third component known as a service discipline. This third component specifies the manner in which service is dispensed to customers who are present at the service facility. For example, customers may be served on a first come first served basis, or in accordance with an externally assigned set of priorities, or on a rotating (i.e., round robin) basis. A number of service disciplines which are important in

the analysis of computer systems will be discussed more thoroughly in later sections of this chapter.

Once a queueing system has been specified by identifying its three primary components, the analysis of the system can begin. As already mentioned, the questions of interest typically concern the number of customers at the facility at any given time, the total amount of time necessary to process particular customers through the system, and the length of the periods during which the service facility is continuously busy serving customers.

Because random factors operate in all queueing systems, the questions of interest can only be answered in terms of random variables or expected values of random variables. As an example of this type of solution, suppose that an initial reference point is established and designated as time zero, and let time t denote the point in time that is t seconds after time zero. Assuming that the number of customers in the system at time zero is known and that the arrival process, the service time distribution and the service discipline are all specified, it is then conceptually possible to calculate $P_n(t)$ - the probability that the number of customers in the system at time t is equal to n - for each value of n (i.e., for n=0,1,2, ...).

In most queueing systems of interest the value of $P_n(t)$ tends to stabilize after an initial period of fluctuation.

That is, the probability distribution characterizing the number of customers in the system eventually becomes invariant with respect to time. Systems which stabilize in this manner are said to become stationary, and the stable distributions which are eventually attained are known as steady state, equilibrium or stationary distributions.

In ergodic systems the final steady state distribution is independent of the state the system starts in at time zero. Thus, a steady state distribution can be used to characterize an ergodic queueing system when all that is known is the arrival process, the service time distribution, the service discipline, and the fact that the system has been in operation for a relatively long period of time.

All the research papers to be discussed in this chapter and the next are directed towards obtaining steady state solutions for ergodic queueing systems. However, it should be noted that it is sometimes possible to obtain time dependent solutions which, in effect, describe the behavior of systems as they progress from some initial state to the equilibrium state. Because of their mathematical complexity and specialized nature, the solutions obtained for the time dependent case have never been directly applied to the analysis of computer systems. Takacs (78) presents a comprehensive account of the known results in this area.

Before closing this section it would be worthwhile to

mention a few modifications of the basic queueing process which are of interest in certain situations. The first of these concerns the number of servers which make up the service facility. The assumption here is that each server is capable of providing service to only one customer at a time. Thus, if there are N customers present at a service facility made up of S servers and N is greater than S, then S customers will be receiving service and N-S customers will be waiting. If N is less than or equal to S, all N customers will be receiving service and no customers will be waiting. Most applications of queueing theory to computers assume S is equal to one, but there are examples such as multiprocessing systems for which some other value of S would be appropriate.

It is important to distinguish the case of multiple servers within a single service facility from the case of queueing networks. In queueing networks there are a number of different service facilities organized so that customers leaving one may proceed to another. Thus, separate queues build up at each service facility in the network. Network parameters include the number of servers present at each facility and the probability that a customer leaving a particular facility will proceed to another specified facility. A number of papers dealing with the theory of queueing networks are discussed in Chapter 3.

Now that the fundamental aspects of queueing theory have been introduced, it is possible to examine some of the

applications of this branch of mathematics to computer systems analysis. Each of the remaining sections of this chapter will focus on one particular area of application.

QUANTUM CONTROLLED SERVICE DISCIPLINES

In interactive time-sharing systems it is usually considered undesirable to keep a short job waiting simply because a substantially longer job has entered the system sometime before it. As a result such systems do not normally process jobs strictly on a first come first served (FCFS) basis. Instead they employ scheduling algorithms which attempt to insure that relatively short jobs do not have to wait in the system for excessively long periods of time.

Scheduling algorithms which provide short jobs with this type of preferential treatment have been the subject of extensive analysis over the past few years. Since most of the algorithms studied belong to the class of quantum controlled service disciplines, it is useful to consider the structure of this class as a whole before examining the behavior of specific algorithms.

The essential feature which characterizes quantum controlled service disciplines is that each job is permitted to run on the system (i.e., the CPU) for a certain period of time known as a quantum. If a job terminates before its quantum has expired, it leaves the system immediately. Otherwise, it returns to the queue of waiting jobs when its quantum expires. In either case, another job is then immediately selected from the queue of waiting jobs and granted the next quantum of CPU processing. The algorithm

continues to operate in this manner so long as there are any jobs in the system waiting for service.

An important feature of quantum controlled service disciplines, in addition to the relative ease with which they can be implemented, is the fact that they can provide preferential treatment to short jobs even though they presume no a priori knowledge of the amount of processing that incoming jobs require. As will be demonstrated in the next section, it is theoretically possible to devise service disciplines which are superior to the quantum controlled type if such a priori information is available. However, because such information is difficult and oftentimes impossible to reliably obtain, designers of interactive time-sharing systems will probably never entirely discard service disciplines of the quantum controlled type.

The mathematical analysis of quantum controlled service disciplines has generated a surprisingly large number of publications. In order to categorize these publications and present them in a relatively coherent manner, the following strategy has been adopted. First, a set of five components which are present in all queueing theoretic models of quantum controlled service disciplines will be identified. Each component will be considered individually, and all the subcategories which have been studied in the literature will be discussed. Then each paper will be classified by specifying the particular sub-category of each component that was

used to construct the model examined in the paper. The final outcome of this procedure is presented in Table 2-1 (pp. 32-33) for a total of twenty-nine papers which were published in the period 1964-1970.

The five components used to make this classification are the scheduling algorithm, the quantum type, the service time distribution, the arrival process and the overhead assumption. These components along with their associated sub-categories are represented schematically in Figure 2-3 (pp. 30-31). The selection of these components was motivated by earlier survey papers prepared by Coffman (18), Estrin and Kleinrock (31), and McKinney (61), and so the material presented here may be regarded as a natural extension of this earlier work.

Scheduling Algorithms

As indicated in Figure 2-3, only two components are required to specify a quantum controlled service discipline: the scheduling algorithm, which determines the order in which jobs are selected for service at the end of each quantum, and the quantum type, which determines the amount of processing time allocated to a job once it has been selected for a quantum of service.

Essentially only two classes of scheduling algorithms have been considered in the literature, round robin (RR) and foreground background (FB). Under the RR discipline jobs entering the system form a single queue in order of arrival.

Each time a new job is to be selected for a quantum of processing, it is taken from the head of the queue. If a job requires additional processing at the end of a quantum, it is placed at the tail of the queue as if it were a new job. Thus, before a job can receive an additional quantum, each job which was present in the system at the end of its previous quantum must first receive a quantum of its own. The operation of such a scheduling algorithm is depicted schematically in Figure 2-1.

Under the FB discipline, jobs entering the system also form a single queue in order of arrival. This queue, which is known as the foreground queue, is served on a FCFS basis with each job being granted one quantum of processing. If a job requires additional processing at the end of its quantum, it does not return to the tail of the foreground queue as in the RR algorithm but instead returns to the tail of the first background queue. After a wait in the first background queue, a job receives its second quantum of processing and then proceeds to the third background queue, then the fourth, and so on until its processing requirement is finally satisfied.

An important feature of FB algorithms is that each time a new job is to be selected for a quantum of processing, it is taken from the head of the highest priority non-empty queue. In this context the foreground queue has highest priority, the first background queue has second highest

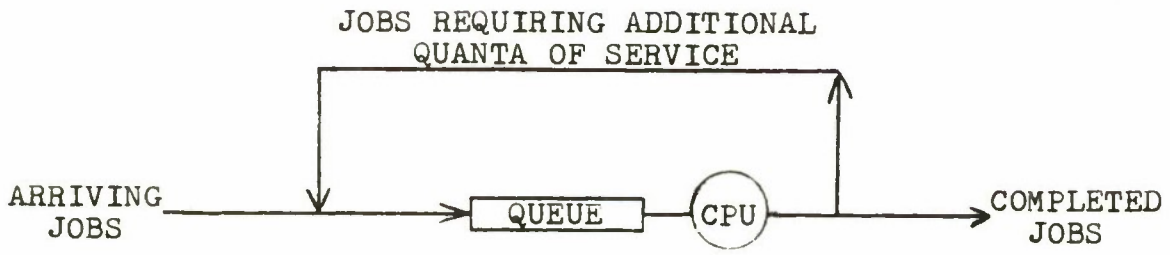


Figure 2-1

The Round Robin (RR) Scheduling Algorithm

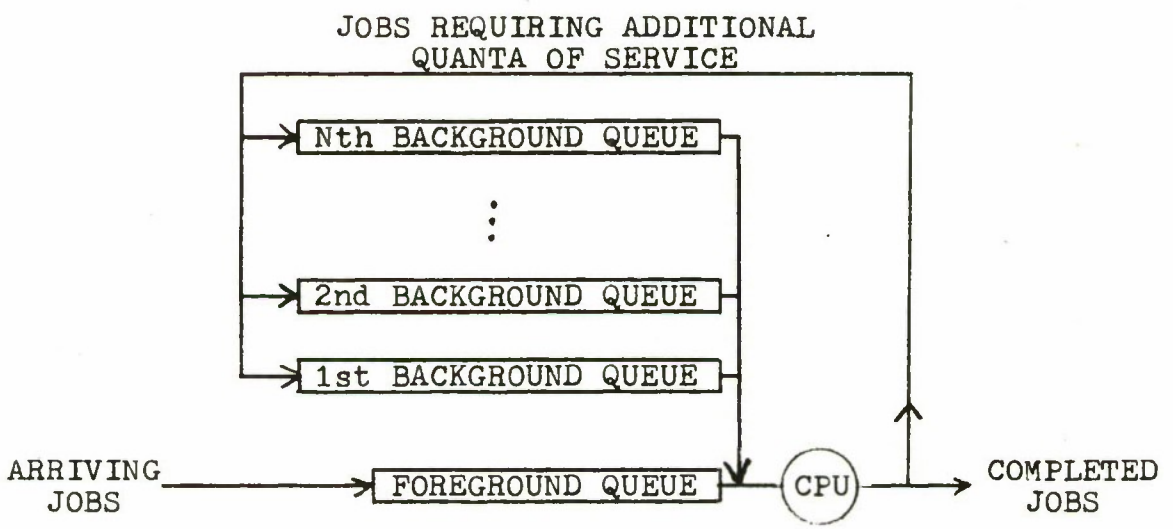


Figure 2-2

The Foreground Background (FB) Scheduling Algorithm

priority, and in general the n^{th} background queue has $n+1^{st}$ highest priority. Within the foreground level and each background level jobs are queued in the order in which they arrive at that particular level and served on a FCFS basis. The entire process is illustrated in Figure 2-2.

To complete the description of the FB algorithm it is necessary to discuss the disposition of jobs which complete a quantum of service on the lowest priority background level but still require additional processing. One fairly common procedure is to continue to give such jobs additional quanta until they finally run to completion. If in the meantime a new job enters the system, that job will begin to receive service as soon as the job being served comes to the end of its next quantum since the new job will be in a higher priority queue. This particular method of managing the lowest priority background queue is known as the quantum controlled first come first served discipline.

A second alternative is to operate the lowest priority background queue under a RR discipline so that a job completing a quantum of service immediately cycles back to the tail of that queue. It is also possible to let jobs in the lowest priority background queue simply run to completion without any possibility of preemption. This third alternative may be thought of as a special case of either of the first two in which the quantum length of the lowest priority queue has become infinite.

Still another way to deal with this problem is to postulate the existence of an infinite number of background levels. Once this is done, the problem disappears entirely since there no longer is a lowest priority queue. While this solution in no way affects the treatment given to short and medium length jobs, it elegantly removes the singularity associated with the lowest priority queue and thus gives the algorithm a more uniform structure.

For notational purposes, FB algorithms incorporating an infinite number of background levels will be identified as FB_{∞} algorithms while FB algorithms incorporating a finite number of background levels will be identified as FB_N algorithms. A specific example with, for example, a total of three levels (two background and one foreground) will be identified as an FB_3 algorithm. In order to keep the notation simple, no attempt will be made to specify the way in which the lowest priority queue is managed in the finite case.

There are a few minor variants of the basic RR and FB scheduling algortihms which have received some attention in the literature. These algorithms will be identified for purposes of this discussion as follows:

RR/D Round Robin with Delayed Entry - Jobs which arrive at the system do not enter the round robin cycle until after some period of time has elapsed. This algorithm might be useful in modeling a system in which the arrival of new jobs is detected by the periodic polling

of a set of flags. Kleinrock (56) applies the label "selfish round robin" (SRR) to a particular subcase of this class, while Krishnamoorthi and Wood (60) refer to another subcase as a schedule queue discipline.

FB/P Foreground Background with Priority Entry - This algorithm is identical to the FB algorithm except that jobs may enter directly at any of the background level queues as well as at the foreground level. The highest priority class jobs enter at the foreground level and jobs of progressively lower priority enter at progressively higher background levels.

FB/PP Foreground Background with Priority Entry and Priority Service - In this modification of the FB/P algorithm, priority class not only determines a job's initial point of entry into the system but also determines the intra-level service order. That is, within each level, higher priority jobs are served first and jobs of the same priority are served on a FCFS basis.

FB/PO Foreground Background with Priority Entry and Oldest Job First Service - In this modification of the FB/P algorithm, jobs within a particular level are served in the order of their initial arrival at the system (i.e., oldest job first) rather than in the order of their arrival at that level (i.e., FCFS).

FB/NQ Foreground Background with Non-Standard Queue Selection - In standard FB algorithms, the foreground queue has

highest priority and the $n^{\underline{th}}$ level background queue has $n+1^{\underline{st}}$ highest priority. FB/NQ algorithms postulate some other priority ordering. However, jobs still enter at the foreground level and work their way up through successive background levels.

Quantum Types

As stated previously, in order to specify a quantum controlled service discipline it is necessary to identify both the scheduling algorithm and the quantum type. When classifying quantum types it is convenient to first make the distinction between deterministic quanta and random quanta. In the deterministic case the length of a quantum is completely determined once a set of associated values known as quantum defining factors is specified. In the random case the quantum defining factors serve only to determine the probability distribution characterizing the length of the associated quantum; quantum length itself is thus a random variable rather than a constant in this case.

If a job terminates before its final quantum expires, it leaves the system immediately and a new job is then allocated the next quantum of processing. Thus, in systems employing deterministic quanta, it is not necessarily true that all quanta corresponding to a given set of quantum defining factors have the same actual length. However, all quanta corresponding to a given set of quantum defining

factors do have the same maximum length.

In the case of random quanta, the random variable associated with a particular set of quantum defining factors may correspond to either the maximum quantum length or the actual quantum length. This additional degree of freedom results from the fact that since quantum size is already a random variable its distribution function can be chosen to reflect the fact that jobs terminate at arbitrary points in time.

In the quantum identification scheme to be used in this discussion, the initial letter will indicate whether the quantum is deterministic (D) or random (R), and the following letters will indicate the quantum defining factors. Using this scheme, seven different quantum types which have appeared in the literature may be identified as follows:

DI Identical length quanta are allocated to all jobs.

RI The identical distribution characterizes quantum length for all jobs.

DP Different classes of jobs (i.e., different priority groups) are identified with each class having its own associated quantum length.

DN Quantum length is defined as a function of the number of quanta a job has already received in an RR system or the level it has attained in an FB system.

RN At each level in an FB system, quantum length is characterized by a particular distribution function.

DPN Quantum length is defined as a function of both the

assigned priority class of a job and the number of quanta the job has already received in an RR system or the level it has attained in an FB system.

DD Quantum length is defined as a function of some dynamic property of the system such as the number of jobs currently present or the number of jobs which have arrived in the recent past.

A number of authors have also investigated deterministic quanta in the limiting case where quantum size approaches zero. The term "processor sharing", which is due to Kleinrock (53), is commonly used to identify this limiting case. The following quantum types have appeared in conjunction with processor sharing systems:

DIZ Limit of type DI quanta as quantum size approaches zero.

DPZ Limit of type DP quanta as quantum size approaches zero.

DPNZ Limit of type DPN quanta as quantum size approaches zero.

Once a quantum controlled service discipline has been defined by specifying a scheduling algorithm and a quantum type, its behavior may be evaluated by any of a number of different methods. For example, it is possible to implement the discipline within an actual system and the make appropriate measurements while the system is operating. Alternatively, it is possible to incorporate the discipline into

a simulation model and then evaluate the model using Monte Carlo techniques. A third possibility is to incorporate the discipline into a mathematical model and then evaluate the model analytically. This third possibility will be examined more closely in the discussion which follows.

The simplest mathematical model which can be applied to the evaluation of quantum controlled service disciplines is probably the single server queue. For purposes of this discussion it is useful to consider such models as being composed of two independent components, a service discipline and a stochastic environment. The first component has already been discussed in considerable detail, and so to complete the description of these models it is only necessary to consider the second component.

A stochastic environment may be defined as everything which must be added to a service discipline in order to completely specify a particular queueing model. More specifically, a stochastic environment consists of an arrival process, a service time distribution and an overhead assumption. The nature of each of these three components will now be considered in some detail.

Overhead Assumptions

Overhead assumptions are needed to specify the amount of time necessary to transfer control of the CPU from one job to another when a quantum expires or a job terminates.

Four overhead assumptions which have been used in conjunction with quantum controlled service disciplines may be identified as follows:

Z Zero Overhead - The CPU is switched from one job to another in zero time.

C Constant Overhead - A fixed amount of time is required to switch the CPU from one job to another.

CPN Constant Overhead for Specific Situations - The amount of time required to switch the CPU from one job to another is some known function of the job's priority class and the number of quanta it has already received in an RR system or the level it has attained in an FB system.

R Random Overhead - The amount of time required to switch the CPU from one job to another is an arbitrarily distributed random variable.

Arrival Processes

In an early paper, Kendall (50) classified a number of arrival processes and service time distributions which are important in the theory of queues. Using an expanded and slightly modified version of Kendall's notation, the arrival processes which have proven useful in the analysis of quantum controlled service disciplines may be identified as follows:

B Bernoulli Arrivals - At the end of each quantum, a Bernoulli trial is made to determine whether or not a

new job is to arrive. The probability of success (i.e., an arrival) is assumed to be the same in each trial. If quantum length is constant (i.e., type DI), Bernoulli arrivals imply geometrically distributed inter-arrival intervals.

M Poisson Arrivals - In any time interval of length T, the probability that there will be exactly k arrivals is equal to $\frac{(aT)^k}{k!} e^{-aT}$ where a is some positive constant. This implies that inter-arrival intervals are exponentially distributed with mean 1/a.

M_f Finite Source Poisson Arrivals - If the number of jobs at the CPU is equal to j, then the amount of time until the next arrival is an exponentially distributed random variable with mean 1/a(N-j) where a is some positive constant and N is an integral constant. No arrivals are possible when the value of j reaches N, and hence queue size is bounded by N.

G General Arrivals - The inter-arrival intervals are entirely arbitrary and possibly correlated random variables. Usually, all that is possible under general arrival assumptions is to state the solution of one problem in terms of the solution of some other problem.

In time-sharing systems, each active terminal functions as a source of jobs (i.e., CPU processing requests). Since a terminal is not normally permitted to generate a new pro-

cessing request until its previous request has been completed, the job arrival rate typically declines as the number of incomplete jobs waiting at the CPU increases. However, this effect becomes less marked as the total number of active terminals increases and, in the limiting case where the number of active terminals approaches infinity, it disappears entirely.

Both Bernoulli and Poisson arrival processes correspond to this limiting case since neither exhibit any correlation between arrival rate and queue length. Hence these processes are sometimes referred to as infinite source arrival processes and are best suited for modeling time-sharing systems with a large number of active terminals. While the Bernoulli arrival process may be somewhat easier to conceptualize because of its discrete nature, both processes are mathematically attractive since both incorporate the memoryless property discussed in Appendix A.

The finite source Poisson arrival process explicitly represents the case in which the arrival rate decreases as the number of jobs already waiting for CPU service increases. This is done by assuming that the length of time between the completion of a job associated with a particular terminal and the generation of the next job by that same terminal is an exponentially distributed random variable with mean 1/a. This random variable, which is commonly referred to as "think time", is assumed to have the same distribution at

all terminals. Then, if the total number of terminals in the system is equal to N and the number of jobs at the CPU is equal to j, it follows that the amount of time until the next arrival is an exponenitally distributed random variable with mean 1/a(N-j). This is the rationale underlying the finite source Poisson arrival process.

Service Time Distributions

As is evident from the preceding discussion, one way of characterizing arrival processes is by defining the distribution of their inter-arrival intervals (i.e., the intervals between the arrival of successive customers). These same distributions are often used to characterize the amount of processing time that individual jobs request, and when this is done the abbreviation used to identify the arrival process is also used to identify the corresponding service time distribution. As the following list indicates, three of the four service time distributions which have been analyzed in the literature exhibit this correspondence.

B Bernoulli Sum Service Times - At the end of each quantum, a Bernoulli trial is conducted to determine whether the job which has just completed the quantum is to leave the system or re-cycle for at least one more quantum of processing. The probability of leaving the system is assumed to be the same in each trial. Thus the total amount of service time required by a

job is distributed in the same manner as the inter-arrival intervals of a Bernoulli arrival process operating with the same quanta.

M Exponential Service Times - The total amount of processing time required by each job is an exponentially distributed random variable. These random variables are all independent and identically distributed.

G General Service Times - The total amount of processing required by each arriving job is an arbitrarily distributed random variable. These random variables are all independent and identically distributed.

H Hyperexponential Service Times - The total amount of processing time required by an arriving job is a hyperexponentially distributed random variable. These random variables are all independent and identically distributed.

In practice, the hypothesis of exponential service times has proven to be a crude but not unacceptable approximation to observed service times. However, Walter and Wallace (82) indicate that a more precise fit to empirical data can be obtained by assuming that service times are hyperexponentially distributed. One way to interpret the hyperexponential assumption is to imagine that there exist two classes of jobs, class A and class B, with incoming jobs falling into class A with probability p_A and into class B with probability p_B ($p_A + p_B = 1$). Jobs in class A

are assumed to have exponentially distributed service times with mean 1/a while jobs in class B are assumed to have exponentially distributed service times with mean 1/b ($a \neq b$). Under these conditions service times will be distributed as $p_A ae^{-at} + p_B be^{-bt}$ which is a hyperexponential density function of the second degree. In interactive time-sharing systems, class A may be associated with editing requests and class B with all other requests.

Summary and Evaluation

In summary, a quantum controlled service discipline is defined by specifying its scheduling algorithm and its quantum type. Once a service discipline has been defined, it may be evaluated by the use of queueing theory. To do this it is necessary to embed the service discipline in a stochastic environment which, as a minimum, must consist of an arrival process, a service time distribution and an overhead assumption. This procedure for model construction is depicted in Figure 2-3.

As indicated in Table 2-1, a large body of published research has been devoted to analyzing models which fall within the framework of Figure 2-3. Most of the papers

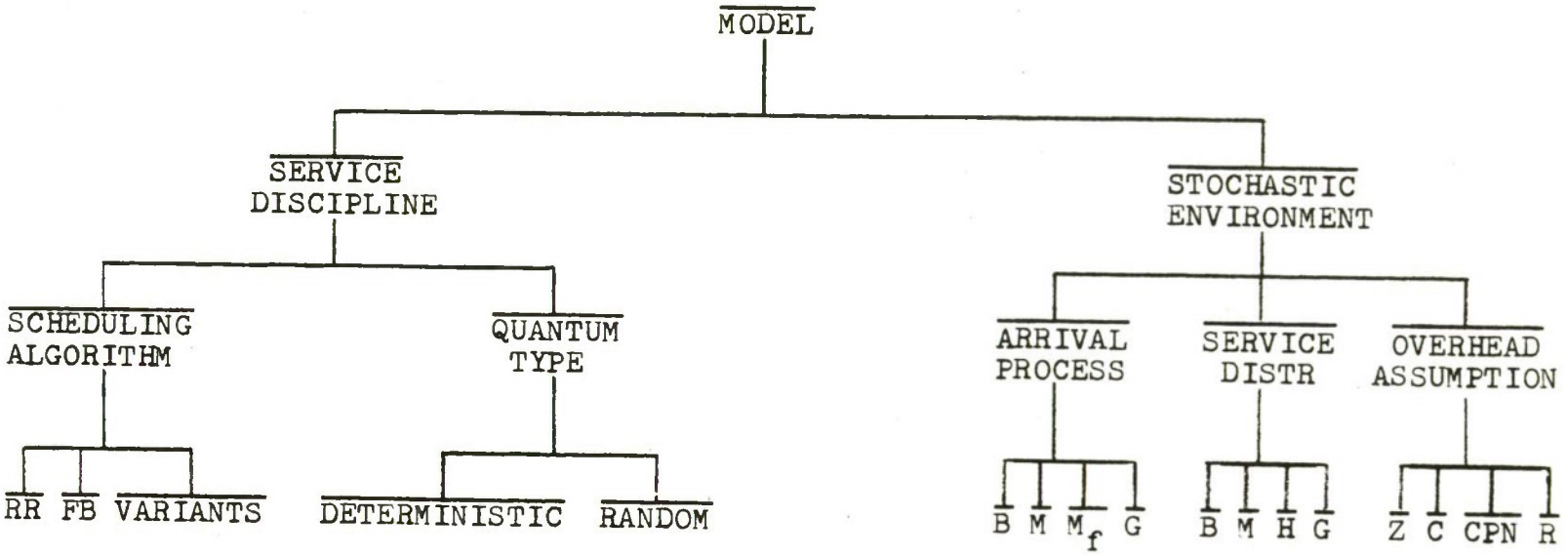


Figure 2-3 Part A

Queueing Models of Quantum Controlled Service Disciplines

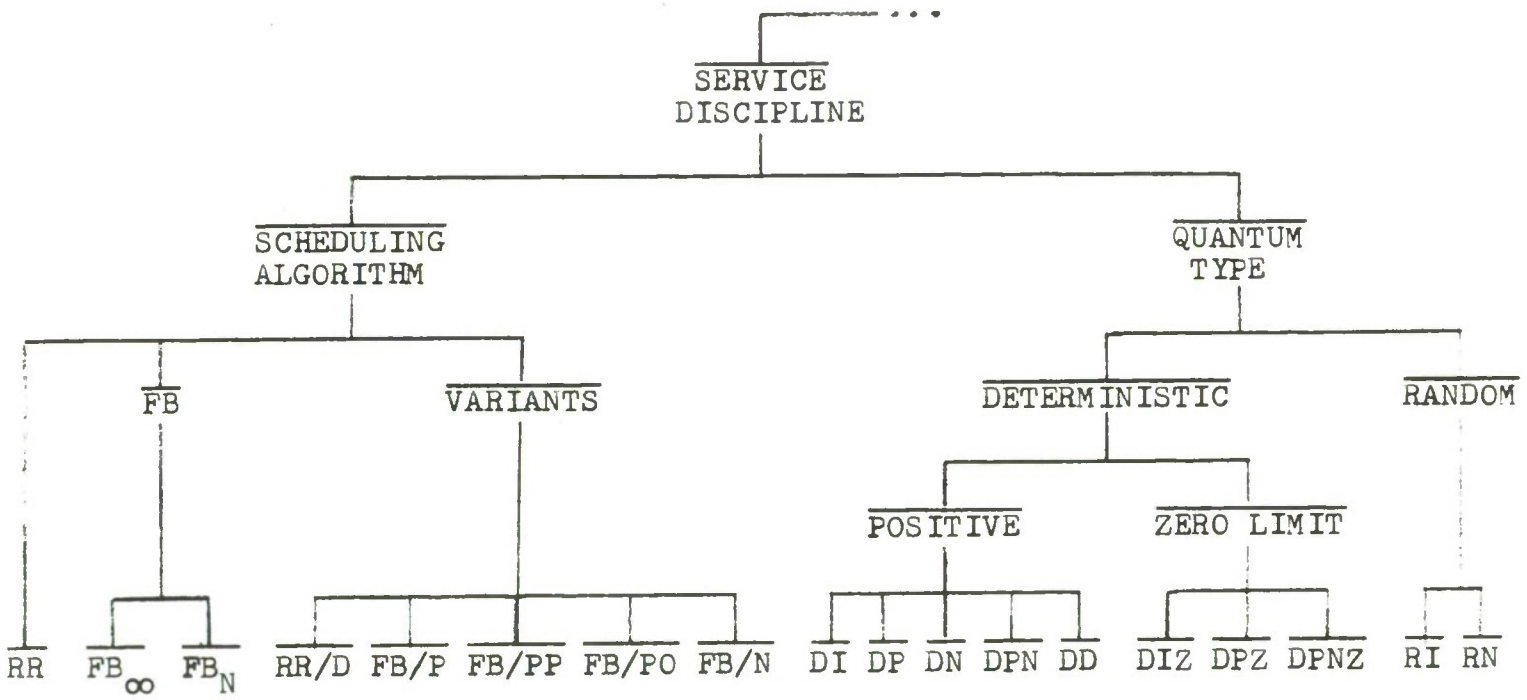


Figure 2-3 Part B

Detailed Structure of the Service Discipline

AUTHOR (Ref. No.), DATE	SCHEDULING ALGORITHM	QUANTUM TYPE	ARRIVAL PROCESS/ SERVICE DISTR.	OVER-HEAD
Adiri & Avi-Itzhak (5),1969	RR	DI	M_f/M	C
Baskett (8),1970	RR	DIZ	M_f/H	Z
Chang (13),1966	RR	RI	M/B	Z,R
Coffman (17),1966	RR,FB_2,FB_N FB_∞/PP	DI,DN,DD DIZ	M/M, M_f/M	Z,C
Coffman (18),1967		--- SURVEY ---		
Coffman (19),1968	RR,FB_2	DI	M/M	C
Coffman (20),1968	RR,FB_2	DD	B/B	Z
Coffman & Kleinrock (22),1968	RR,FB_N FB_∞/P	DI,DIZ DPZ	B/B, M/M	Z
Coffman & Krishna-moorthi (23),1964	RR,RR/D	DI	M_f/B	C
Coffman & Muntz (24),1969	RR,FB_∞	DIZ	M/G	Z
Coffman, Muntz & Trotter (25),1970	RR	DIZ	M/M	Z
Estrin & Kleinrock (31),1967		--- SURVEY ---		
Fife (34),1966	FB_3/N	DN	M_f/H	C
Greenberger (44), 1966	RR	DI	M_f/M	C
Kleinrock (51),1964	RR	DI	B/B	Z

Table 2-1 Part A

Research and Survey Papers Dealing with the Analysis of Quantum Controlled Service Disciplines

AUTHOR (Ref. No.), DATE	SCHEDULING ALGORITHM	QUANTUM TYPE	ARRIVAL PROCESS/ SERVICE DISTR.	OVER-HEAD
Kleinrock (53),1967	RR	DIZ,DPZ	M/M	Z
Kleinrock (54),1968	RR	DIZ	M_f/M	Z
Kleinrock (55),1969	ALL TYPES	DPN	G/G	CPN
Kleinrock (56),1970	RR/D	DIZ	M/M	Z
Kleinrock & Coffman (57),1967	ALL TYPES	DPN DPNZ	G/G	Z
Krishnamoorthi (59),1966	RR	DI	M_f/M	C
Krishnamoorthi & Wood (60),1966	RR,RR/D	DI	M_f/M	C
McKinney (61),1969		--- SURVEY ---		
Patel (64),1964	RR,FB_∞/PO	DI,DN	M_f/G, M/G	Z,C
Rasch (66),1970	RR	DI	M/M	Z,C
Sakata, Noguchi & Oizumi (68),1969	RR	DIZ	M/G	Z
Scherr (69),1965	RR	DIZ	M_f/M	Z
Schrage (70),1967	FB_∞	RN,DN DIZ	M/G, M/M	Z
Shemer (75),1967	RR,FB_∞/P	DI,DN	M/M	Z

Table 2-1 Part B

Research and Survey Papers Dealing with the Analysis of Quantum Controlled Service Disciplines

present an analysis of a specific model: that is, of a particular service discipline operating in a particular stochastic environment. Usually the point of the analysis is to obtain an algebraic formula which expresses the expected waiting time of a job as a function of the job's execution time. This makes it possible to determine the extent to which short jobs are favored over long jobs. In addition, since these formulas typically include quantum length, overhead time, arrival rate and mean service time as parameters, it is possible to examine the treatment of long and short jobs under a wide variety of conditions.

There are a number of uses to which such formulas might be put. Perhaps the most obvious is the determination of optimal quantum length for models in which both the quantum length and the overhead time are non-zero. In such models longer quanta reduce overhead but also increase the expected waiting time for short jobs. On the other hand, very short quanta result in a high percentage of overhead, thus increasing the expected waiting time for all jobs including the short ones. Hence it is reasonable to suppose that an optimal quantum length exists at some intermediate point.

To determine this optimal length it is first necessary to define exactly what it is that is being optimized. A convenient way to approach this problem is to define a cost function which reflects the delays associated with the system and then to try to minimize cost. Since quantum controlled

service disciplines are primarily designed to provide good service to short jobs, it is reasonable to assume that the cost associated with keeping a short job waiting is greater than the cost associated with keeping a long job waiting. It is also reasonable to assume that the longer a job is kept waiting, the greater the cost.

The simplest way to represent these two assumptions mathematically is to define the cost of keeping a job with total service time S waiting for a period of time T as $T \cdot F(S)$ where F is a positive, non-increasing function of S. Cost functions constructed in this manner have been analyzed by Fife (34), by Greenberger (44), and most comprehensively by Rasch (66). More information must be collected before any general results can be reported, but the potential for additional work in this area is quite promising since a number of models have been solved analytically and only a very few have been optimized with respect to quantum length.

A number of other significant problems in the area of quantum controlled service disciplines also exist. For example, it would be valuable to compare the performance of RR and FB algorithms under a variety of overhead assumptions to determine the optimal algorithm for a specific application. The optimal choice could be specified under relatively simple quantum assumptions (e.g., DI) or under more complex quantum assumptions (e.g., DPN or DD). Note that it is necessary to solve the optimal quantum length problem for

each specific algorithm before attacking this comparison of optima problem. Fife (34) deserves recognition for the results he has obtained in this area, but unfortunately his work is of somewhat limited appeal because of its numerical rather than algebraic nature; all the other papers appearing in Table 2-1 are distinctly algebraic.

Another consideration which naturally emerges from this discussion is the relative importance of the arrival process and the service time distribution in optimization problems in general. For example, do exponential and hyperexponential service time distributions or finite and infinite source arrival processes yield different optima? If so it is necessary to examine the arrival and service time statistics very carefully before selecting an algorithm. If not, it may be possible to discover general guidelines which can be followed with confidence in a variety of situations.

It should be clear from the preceding discussion that even though a considerable amount of effort has already been expended analyzing quantum controlled service disciplines, many problems remain unsolved. However, these problems are qualitatively different from the earlier ones in that they deal with the optimization and comparative evaluation of proviously analyzed models rather than the determination of conditional waiting times for newly proposed models. Thus solutions to these new problems will be built upon existing knowledge in a way that is characterictic of many other branches of science and mathematics.

CONVENTIONAL PRIORITY DISCIPLINES

In most real-time systems, incoming jobs are assigned to different priority groups according to the relative urgency with which they must be completed. Jobs which must be completed in the shortest possible time are assigned to the highest priority group, slightly less urgent jobs are assigned to the next highest priority group, and so on. Note that a job's priority group, which is the basis for providing preferential service, is specified at job entry time. This is in contrast with interactive time-sharing systems employing quantum controlled service disciplines since these systems use total running time as the basis for providing preferential service even though this factor is not assumed to be specified at job entry time.

The service disciplines used in formulating queueing theoretic models of real-time systems are known as conventional priority disciplines. These service disciplines all exhibit the following three characteristics:

1. Jobs are assigned to priority groups at the time they first enter the system.
2. Whenever a server becomes available it is always assigned to a job from the highest non-empty priority group.
3. A job's priority group never changes.

Conventional priority disciplines may be partitioned into a number of subcases by giving different interpretations to the notion of availability which appears in Characteristic 2. For example, if a server becomes available only after completing the processing of a job, the discipline is known as non-preemptive or head-of-line. Under this discipline, a job is always allowed to run to completion even though a higher priority job may arrive while it is being processed.

A second possibility is to assume that servers are always available. Thus, if a higher priority job arrives while a lower priority job is being served, the latter will be ejected from the service facility and the higher priority job will begin to receive service immediately. Disciplines in which jobs can be ejected from the server in this way are known as preemptive disciplines.

Preemptive disciplines may be further subdivided on the basis of the treatment given to jobs which return to the service facility after having been preempted. If such jobs are permitted to simply continue service from the point where they were interrupted, the service discipline is known as a preemptive-resume discipline. If, on the other hand, jobs are forced to return to their original starting points and repeat everything they have already done, the discipline is known as preemptive-repeat-identical. This discipline is of some interest in computer applications where it may be

preferable to restart a job from its initial point rather than save all its status information and temporary storage so it can be started again from the point of interruption.

Preemptive-repeat-identical disciplines are distinguished from preemptive-repeat-different disciplines by the fact that, in the latter case, the amount of processing time that a job requests is re-calculated each time a job is restarted from its initial point after a preemption. Preemptive-repeat-different disciplines appear to have few if any applications in computer systems analysis and are mentioned only for the sake of completeness.

An early example of the application of non-preemptive priority disciplines to the analysis of computer systems is presented by Chang and Wong (16). In a more recent paper, Chang (15) presents a number of other examples involving both non-preemptive and preemptive-resume disciplines. This latter paper is also valuable for tutorial purposes since a number of analytic techniques are carefully reviewed.

The discussion thus far has assumed that within each priority group service is provided on a FCFS basis. It is also possible to study conventional priority systems in which the intra-group service discipline is of the quantum controlled type. For example, Chang (14) has analyzed a model in which each priority group functions as a type RR, RI, M/B, Z system, and Adiri (3), (4) has considered the case in which each priority group functions as a type RR,

DI, M/M, C system.*

Both Chang (14) and Adiri (3) assume that the priority discipline operating between groups is of the preemptive-resume type. In his more recent paper, Adiri (4) considers three additional inter-group disciplines. The first assumes that once a job has been allocated a quantum of processing, no preemption is possible until the end of that quantum is reached. Preemption occuring at the end of a quantum is of the preemptive-resume type.

In the second discipline jobs may be preempted at any time, but preemption causes the intermediate results developed during the current quantum to be lost. That is, a preempted job is restarted from the beginning of the quantum it was receiving when it was interrupted. This discipline is thus midway between the preemptive-repeat-identical discipline and the preemptive-resume discipline.

The third discipline is a combination of the second discipline and the standard preemptive-resume discipline. Preemption is permitted at any time and, if a job is preempted during the initial overhead phase (i.e., the set-up period) of the quantum, the quantum is considered lost and the job is later restarted from the beginning of that quantum as in the previous case. However, if a job is interrupted during the processing phase of the quantum, the job

*These classifications correspond to the column designations of Table 2-1.

is later restarted from the point of interruption as in the preemptive-resume case. This particular discipline would appear to be a logical choice for actual real-time systems employing quantum controlled service disciplines within each priority group.

Schrage (71) presents a model which is similar to Adiri's except that the service discipline within each priority group is of the FCFS type. Schrage assumes that each job is partitioned into a set of non-preemptive, preemptive-resume and preemptive-repeat intervals. A higher priority job arriving during a non-preemptive interval must wait until the end of that interval before gaining control of the CPU which is finally relinquished on a preemptive-resume basis. During preemptive-repeat intervals, higher priority jobs immediately gain control of the CPU, and the job which was preempted is forced to begin again from the start of the preemptive-repeat interval. This corresponds to Adiri's second discipline. Finally, during a preemptive-resume interval, preemption occurs in the normal preemptive-resume sense.

Schrage also treats the possibility of overhead during each preemption. His model is thus capable of representing actual systems with a high degree of precision. Unfortunately, examples including non-zero preemptive overhead prove difficult to treat analytically and are only discussed in numerical terms.

Since the primary reason for implementing conventional priority service disciplines is to provide higher priority jobs with preferential service, most analyses are concerned with obtaining waiting times for jobs in each priority group. In general, preemptive disciplines favor high priority jobs more than non-preemptive disciplines, but the optimal amount of preemption to permit is an open question, especially in cases where preemption introduces overhead. Schrage (71) has obtained some preliminary results along these lines, but the potential for additional work in this area is great.

While not directly related to the primary concerns of this section, it is interesting to note that priority disciplines can be used to analyze limiting aspects of the rationale which underlies the quantum controlled service disciplines discussed in the preceding section. Recall that the primary purpose of quantum controlled service disciplines is to provide short jobs with preferential service. The problem is that job length is not assumed to be specified at the time a job enters the system. However, if such information were available, it would be a simple matter to define a service discipline which, at any point in time, always provided service to the shortest job present.

Such a discipline can be considered to be a conventional preemptive-resume priority discipline where the priority of an entering job is given by the total processing time of that job, shorter jobs having higher priority. If the

arrival process is Poisson with mean rate u and the service time distribution has density function g(t), then there will be a continuum of priority groups corresponding to all the positive real numbers. The arrival process for all the priority groups between r and r' will be Poisson with mean rate $\int_r^{r'} u \cdot g(t)\, dt$, and of course all programs arriving at priority group r will have execution time equal to r. This particular discipline is closely related to the disciplines studied by Phipps (65) and by Schrage and Miller (72).

ROTATING STORAGE SERVICE DISCIPLINES

A great many computer systems utilize rotating disks or drums as auxiliary storage devices. For purposes of this discussion a drum will be defined as any rotating storage device with fixed read/write heads while a disk will be defined as any rotating storage device with movable read/write head. Note that under these conventions, devices commonly known as fixed-head or head-per-track disks are classified as drums.

When analyzing the performance of disks and drums, it is natural to think in terms of a queueing process in which incoming read and write requests represent customers, the disk or drum represents the server, and the amount of time necessary to complete a read or write request represents the service time. The service time in drum systems is the sum of two components, the rotational delay associated with bringing the proper drum sector to the read/write heads plus the actual time required to make the transfer. With disks the service time is the sum of these two components plus the time necessary to move the read/write heads into proper position (i.e., the seek time). Since service times are slightly simpler in the case of drums, these devices will be considered first.

From an analysis standpoint, the most interesting aspect of drum systems is the order in which transfer requests are

serviced. Note that the FCFS discipline is highly inefficient since the time spent waiting for a particular sector to rotate into position under the drum heads could be better spent transferring data to or from the sectors passing under the heads during the waiting period. A more efficient discipline can thus be constructed by sorting all requests according to the drum locations they reference and then always servicing the request which the heads will reach next. This discipline, which Denning (28) designates as shortest access time first (SATF),* has been studied by Weingarten (84), Denning (28), Coffman (21), and Abate and Dubner (1).

The first three authors consider the case in which transfer requests always reference data blocks of fixed length. Denning is primarily concerned with estimating average service time as a function of the number of requests waiting in the queue. Weingarten and Coffman hypothesize Poisson arrivals and then evaluate system performance as a function of the mean arrival rate, with Weingarten's solution serving as an upper bound to the more exact solution obtained by Coffman. Abate and Dubner, who deal with the case of variable block size, present only approximate results for this more difficult problem.

Since disks are also rotating devices, they too may employ SATF service disciplines. That is, after the heads

*Weingarten (84), for entirely frivolous reasons, refers to this service discipline as the Eschenbach scheme in memory of the Bavarian poet Wolfram von Eschenbach (1170-1220). The term has gained little currency.

have been positioned in a particular location (i.e., on a particular cylinder), the SATF discipline can be used to sequence through all the transfer requests which reference that cylinder. Weingarten (85) has analyzed the effect that such a discipline can be expected to have on system performance. Most other investigators have simply assumed a FCFS discipline within each cylinder, arguing that since the number of cylinders per disk is large - typically on the order of 100 or 200 - it is unlikely that there will be a significant number of requests queued for a particular cylinder at any given time unless disk use is extremely heavy. These investigators thus concentrate on other aspects of disk behavior in their models.

One frequently studied problem concerns the scheduling of transfer requests in a way that reduces disk head movement. Again, the simplest policy is to service all requests on a FCFS basis with no consideration of the resulting head movement. A second policy, termed shortest seek time first (SSTF) by Denning (28), corresponds to the SATF policy in that the heads are moved as little as possible on each seek. This is essentially a step-by-step or local minimization process. Frank (39) considers the global minimization problem of finding the seek pattern which minimizes the total amount of seek time necessary to service all requests present in the disk queue at a given time.

During periods of heavy load both minimization policies

have the potential disadvantage of creating excessively long delays for certain requests. This is because minimizing head movement tends to keep the heads in a particular region for a long period of time. Newly arriving requests directed towards this region will receive good service since they will require comparatively short seeks. However, requests directed towards more distant regions will continue to wait since it would be sub-optimal to move the heads a large distance to serve these requests and then to move the heads all the way back to serve the newly arrived requests.

To avoid the possibility of these excessive delays, Denning (28) has proposed a head movement policy called SCAN in which the heads continually sweep back and forth across the disk, servicing requests for each cylinder they pass but never changing direction until the end of the sweep. Denning (28) compares this policy with FCFS and SSTF and concludes that SCAN is the most desirable even though SSTF is more efficient. Weingarten (85), Sharma (74), and Frank (39) have also chosen versions of SCAN for their main analyses.

Another way of reducing seek time on disks is to organize the data so that the most frequently referenced records are on the middle cylinders. Thus, the heads will never have to move more than half the full seek distance to reach these records. The possible effects of such a policy on performance are considered by Frank (39) and by Abate, Dubner and Weinberg (2). Sharma (74) also discusses the

problems associated with allocation of data on disks, concentrating on situations where a single record may be quite large.

An entirely different analysis problem is associated with disk systems which contain a number of independently positionable sets of heads. Usually, each set of heads serves a different disk drive, with a number of drives connected to a single channel capable of transferring data to or from only one drive at a time. Thus, seek operations can be carried out in parallel while transfers must be processed serially. The performance of systems of this type has been studied by Fife and Smith (36), Seaman, Lind and Wilson (73), and Abate, Dubner and Weinberg (2).

CAPACITY PROBLEMS

The primary emphasis of the preceding three sections has been on the determination of response times and waiting times in various queueing situations. In addition to these time oriented problems, queueing models can also be used to treat a number of space oriented problems related to such issues as the amount of storage required by waiting jobs, the time between queue overflow in finite capacity systems, and the allocation of queue space in finite capacity systems with priority or multiple source inputs. These questions, which will be grouped together under the heading of "Capacity Problems", have been studied in computer oriented contexts by Boudreau and Kac (9), Harrison (45), Weingarten (83), Chang and Wong (16), Chang (13), and Bowdon (10).

Several of these papers treat systems in which arrivals occur in groups or batches. In such systems a random variable is associated with each arrival to specify the number of customers who have just arrived. Inter-arrival intervals are determined by the arrival process as in ordinary queueing systems. Thus ordinary queueing systems can be regarded as special cases of batch arrival systems in which the number of customers per arrival is always equal to one.

A very early example of a batch arrival model is presented by Boudreau and Kac (9). This model consists of an input generator which presents a processing unit with a set

of transactions to process once every n seconds. The number of transactions presented is an integer valued random variable with a positive probability of being equal to zero, and the amount of time required to process an individual transaction is equal to 2n (i.e., twice the input generation period). Under these assumptions Boudreau and Kac calculate the distribution of the number of transactions in the system at equilibrium using a Markov chain approach. They also calculate the average time between queue overflow in cases where only a finite amount of storage is available.

Weingarten (83) presents a model of a message switching computer which also includes batch arrivals. In this model there are n input lines all sending messages to a single processor. The function of the processor is to simply re-transmit these messages on a character by character basis. The amount of time required to re-transmit a single character is assumed to be a constant, and so the amount of time required to re-transmit a message is proportional to the number of characters in the message.

Each input line is assumed to generate messages according to a Poisson process, and the amount of time required to re-transmit each message is assumed to be an exponentially distributed random variable. Weingarten then considers the queueing system in which characters correspond to customers, arrivals occur in batches of exponential size, and the service time per customer is constant. Note that this model

requires a slight modification of the notion of batch arrivals since the number of customers per batch is no longer necessarily integral.

The advantage of Weingarten's model is that the number of characters in the system at any time is simply equal to the waiting time for an arriving message multiplied by the number of characters which can be transmitted per unit time. The waiting time for an arriving message can then be calculated for the simpler case of Poisson arrivals and exponential service times. Weingarten goes on to calculate the average time between queue overflow in cases where only a finite amount of storage is available. Since this problem is more difficult than the corresponding one considered by Boudreau and Kac, a number of simplifications have to be made and only approximate results are obtained.

Harrison (45) considers a message switching computer similar to the one analyzed by Weingarten, the difference being that Harrison defines capacity in terms of stored messages rather than stored characters. Aside from this, Harrison's model closely resembles Weingarten's in that a number of input lines send messages to a single processor for re-transmission, each input line functions as a Poisson source, and message re-transmission times are exponentially distributed. Harrison compares the performance of systems having a dedicated fixed buffer for each line with the performance of systems having a single shared buffer serving

all lines. Needless to say, the latter arrangement proves more efficient. However, overhead factors associated with implementation are not taken into account.

In the batch arrival models presented by Boudreau and Kac and by Weingarten, the service time for each customer is defined as a constant. It is also possible to construct batch arrival models in which the service time for each customer is a random variable. Delbrouck (27) presents a model of this type in which service time is a complex random function including both initial set-up time and processing time. Delbrouck's model is used to analyze certain problems related to the polling of input lines in computer systems. Since the model is not primarily concerned with capacity problems, it will not be discussed further in this section.

Chang and Wong (16, p. 587) present a slightly different approach to the problem of queue capacity. This approach, which is discussed more thoroughly in a later paper by Chang (13, p. 122), is to assume that the amount of space required to store each job is specified by a random variable with generating function $F(z)$. Then, if the generating function for the number of jobs in the queue is given by $U(z)$, the total storage requirement for all jobs in the queue will have generating function $U(F(z))$. This follows directly from the discussion of compound generating functions presented by Feller (32, p. 268). The advantage of this approach is that $U(z)$ can often be determined using standard

procedures or well known formulas. Then, once F(z) is specified, the generating function for the total storage required by the jobs in the queue can be immediately obtained.

Bowdon (10) examines the problem of managing finite capacity queues in systems which incorporate conventional priority service disciplines. The specific model Bowdon considers consists of R priority classes being served on a non-preemptive basis by a single service facility composed of K identical servers. Arrivals to each priority class are generated by a Poisson process, and all jobs have exponentially distributed service times with the same mean.

Bowdon adds to this rather familiar system the additional restriction that queue length may not exceed M jobs. If a new job arrives when the queue length is equal to M, that job is permitted to enter the queue only if there is a job already waiting in the queue whose priority is lower than that of the newly arriving job. In such cases the newly arriving job displaces the lower priority job and the latter is simply considered lost. Likewise, jobs which arrive at a time when the queue length is equal to M and which find no jobs of lower priority already waiting in the queue are also lost. Bowdon then calculates the average number of jobs of each priority class in the queue, the average waiting time for jobs of each priority class, and the probability that a job of a particular priority class will not be displaced by a higher priority job while it is waiting.

NETWORK MODELS

All the examples discussed thus far have been concerned with the analysis of individual processing elements such as CPU's, disks or drums. In actual computer systems these processing elements interact with each other since the completion of a CPU processing request is usually followed by the initiation of an I/O processing request and vice versa. Queueing networks provide a natural mathematical tool for analyzing situations of this type, and so it is not surprising that network models have been applied to a number of problems in computer systems analysis.

As indicated in Chapter 3, a great deal of effort has been devoted to obtaining closed form expressions for the equilibrium distributions of various queueing networks. These distributions may also be obtained numerically using a fairly simple iterative technique once the network parameters are specified. A computer program based on this iterative technique has been developed at the University of Michigan and is described by Wallace and Rosenberg (81). This program has been applied to a number of problems in computer systems analysis as indicated by the work of Fife and Rosenberg (35), Smith (76), (77), and Wallace and Mason (80).

Fife and Rosenberg consider a situation in which programs arrive at the system, are loaded into core through a

single I/O channel, execute for a certain period of time, and then leave. The loading of one program may proceed in parallel with the execution of another so long as there is enough room in core to load the new program. To simplify matters, it is assumed that all programs are approximately the same size and that the capacity of core is five programs.

Arrivals to the system are assumed to form a Poisson process, and both program loading and execution times are assumed to be exponentially distributed. In this model, program execution time includes output transmission time as well as CPU processing time. Since output transmission occurs at teletype speed and can be performed in parallel on a multiplexed basis, and since CPU processing time is quite small in comparison to output transmission time, it is approximately true that program execution proceeds in parallel for all programs in core. Note that this is not identical to the processor sharing assumption discussed previously since parallel operation does not result in a proportional decrease in the execution rate of each job.

Fife and Rosenberg use this model to explore the effect on system performance of the program loading time to program execution time ratio. Since the model itself can be criticized on a number of grounds, especially with the advantage of seven years' hindsight, particular results will not be discussed in detail. The real significance of this work is that it demonstrated, at a relatively early date, the ap-

plicability of numerically evaluated queueing network models to problems in computer systems analysis.

Smith (76) also appears to be more interested in demonstrating the applicability of the method than in attacking a particular problem. Smith's paper actually presents two different models: a highly complex and realistic one which could in principle be analyzed using the queueing network approach, and then a somewhat simplified but still realistic model which is analyzed numerically. The second model consists of a CPU, a disk storage device, and a finite number of interactive user terminals. Programs are loaded from the disk and then request a certain amount of CPU processing. Each time a program completes a CPU request, one of three alternatives is selected: with probability q, a request is made for user terminal input; with probability p, a request is made to load an overlay segment from the disk; with probability r = 1-p-q, the program terminates and a request is made to load the next program from disk. The values of p, q and r are constant and thus do not depend on the number of CPU requests a program has previously made.

Since overlay and new program loading requests both utilize the same I/O device and channel, it is necessary to specify the order in which these requests are served. Smith evaluates two different service disciplines, non-preemptive priority with program loading requests favored and preemptive-repeat priority with overlay requests favored. Pre-

emption is of interest in the second case because it is assumed that program loading requires an average of two seconds while segment overlay requires an average of only 150 milliseconds.

As an additional point of interest, Smith introduces two possible distributions for program loading time, exponential and second order Erlang. This illustrates the point that the servers in numerically evaluated queueing networks need not all be exponential. In fact, any distribution whose Laplace transform is equal to the quotient of two polynomials may be used. Such non-exponential servers are constructed using Erlang's method of stages as described by Wallace and Rosenberg (81). Cox and Smith (26, pp. 110-117) present a more detailed discussion of this method.

Smith's paper thus introduces three important aspects of queueing network models which do not appear in the earlier work of Fife and Rosenberg. These are the use of non-exponential servers, flexibility in service disciplines, and the ability of a customer leaving one server to select the next server according to a fixed set of probabilities.

In a subsequent paper, Smith (77) introduces still another aspect of queueing network models, namely the possibility of having service time depend on queue length. This is useful when modeling drum behavior since, assuming the drum discipline is SATF, the mean time between transfers - and hence the mean service time - will decrease as the

length of the drum queue increases. In addition to an SATF drum, Smith's model contains a CPU and a finite capacity paged core memory. Programs execute on the CPU until they generate a page fault, then request a page transfer from the drum, then execute until the next page fault, and so on until they terminate.

Smith explicitly represents the fact that the amount of time a program executes between page faults depends on the number of pages the program already has in core. This is done by assuming that each time a program requests CPU service, the amount of service provided is an exponentially distributed random variable whose mean is an empirically determined function of the number of pages the program has in core. Smith uses the same approach in his drum model, letting the drum service time be an exponentially distributed random variable whose mean is a function of the length of the drum queue.

Smith then examines system performance under a variety of job mixes, multiprogramming allocation policies, and drum speeds, reaching the conclusion that demand paging in systems with a small amount of main memory and a conventional speed drum results in excessive page traffic and low CPU utilization. In addition to introducing the notion of state dependent service times, this model is noteworthy for its representational fidelity and for the accuracy of its predictions.

Wallace and Mason (80) analyze a paging system that is

quite similar to Smith's, the primary difference being that jobs are initiated by the loading of an entire block of pages rather than by the loading of a single page. The initial block of pages is not intended to represent a pre-paging operation, but instead represents an initial burst of page demands generated during the first millisecond or so of CPU execution. Since the CPU execution time is negligible, this burst of individual requests is combined into the initial block to simplify the model.

After the initial block of pages has been loaded, programs are assumed to attain a kind of equilibrium in which the interval between page faults is no longer sensitive to the total number of pages in core. This represents a simplification of Smith's earlier model. In addition, drum service times are assumed to be insensitive to the length of the drum queue, thus further simplifying Smith's model. Both drum service times and CPU service times (i.e., intervals between page faults) are exponentially distributed.

To complete their model, Wallace and Mason assume that each time a program completes a CPU service request, the decision as to whether a page fault is to be generated or the program is to be terminated is determined by an independent Bernoulli trial. Thus, programs enter the system with an initial block of pages, then alternate between CPU processing and I/O processing for a period of time, and then finally leave.

In this model the size of memory is proportional to the maximum number of programs which may be either queued for or receiving CPU or I/O processing. This number is varied to determine its effect on system performance. It is of interest to note that for a wide range of parameters Wallace and Mason find that there is comparatively little gain in system performance after memory size goes beyond eight programs.

Kleinrock (52) presents a network model which is not based on the numeric approach of the four preceding papers but is instead based on an analytic closed form solution obtained by Koenigsberg (58). Kleinrock first considers a system composed of two processors, Processor 1 and Processor 2. Jobs entering the system are first served by Processor 1, then held in an inter-processor buffer, then served by Processor 2, and then ejected from the system. It is assumed that the inter-processor buffer has a maximum capacity of N jobs.

When the inter-processor buffer fills up, Processor 1 is forced to stop operating and system performance suffers. Since the probability of the buffer filling up depends on N and on the ratio of the service times of the two processors, Kleinrock systematically varies these parameters and calculates the effect on system performance. It should be noted that service times are assumed to be exponential in all cases.

Kleinrock then generalizes the problem by considering the case of M processors with jobs proceeding sequentially from Processor 1 to Processor M and then leaving the system. Again, each inter-processor buffer is assumed to have a capacity of N jobs. Only approximate results are obtained for this more difficult case.

Gaver (40) presents an analytic solution for a rather different network model. Gaver's model consists of a CPU, a fixed number of identical I/O processors, and a fixed number of programs. Each time a program completes a CPU processing request, it generates an I/O processing request and vice versa. All I/O processing requests are exponentially distributed with the same mean, and there is no queueing for I/O requests unless the number of programs currently requesting I/O exceeds the number of available I/O processors. In particular, if the number of programs in the system is less than or equal to the number of I/O processors in the system, queueing for I/O never occurs.

Gaver proceeds to calculate CPU utilization as a function of the number of programs in the system. The unique significance of this work is that Gaver is able to carry out this calculation without explicitly specifying the CPU service time distribution. That is, the CPU service time distribution appears as a parameter in the final expression for CPU utilization. It is thus possible to evaluate the effects of different service time distributions in an effi-

cient manner. As an application, Gaver shows the extent to which CPU utilization is reduced as the variance of the service time increases in cases where the mean service time is held constant.

It is worthwhile to point out the correspondence between Gaver's model and the finite source time-sharing models discussed earlier. As long as the number of I/O processors is equal to the number of circulating programs, Gaver's model corresponds precisely to a finite source time-sharing model with exponential "think times", an FCFS service discipline, and general service times.

The special advantages of queueing network models in computer systems analysis should by now be apparent. Such models are well suited for representing concurrent operation of a number of processing units, and the size of main memory is often representable as the number of circulating customers (i.e., programs) in either some part of or all of the network. In addition, the opportunity exists for realistically representing program behavior as an alternating sequence of CPU and I/O processing requests.

A number of recently developed network models are discussed in subsequent chapters of this thesis. These models include the work of Baskett (8), Arora and Gallo (7), Moore (62) and Tanaka (79). Baskett's model is discussed on pages 189-192 of Chapter 7 while the other three models are discussed at the beginning of Chapter 8.

CHAPTER 3: SURVEY OF QUEUEING NETWORK RESEARCH

EARLY DEVELOPMENTS

Output Distributions

A queueing network is a collection of service facilities arranged in such a way that customers must proceed from one to another in order to fulfill their service requirements. The essential feature of such systems is that the output of one service facility may make up part or all of the input to another service facility. Thus some of the early research in this area was devoted to determining the distribution of the output of a single service facility.

Burke (11) studied the case of a single service facility composed of an arbitrary number of parallel exponential servers. Under the assumption of Poisson arrivals, Burke proved that the steady state departure process is of the same form as the arrival process (i.e., Poisson). He also showed that the departure process is independent of the queue size left by a departing customer. Reich (67) showed Burke's first result is not true in general by constructing a specific example in which the arrival and departure processes differ at equilibrium. Finch (38) examined the generality of Burke's second result and was able to show the the departure process is independent of the queue size left by a departing customer if and only if service times are exponential and infinite length queues are permitted.

With the work of Reich and Finch posing potential complications to any general analysis, most studies of queueing networks have been restricted to the case in which individual service times are exponential and arrivals to the network, if any, are Poisson. In this case the entire system can be treated as a continuous time Markov process and the steady state distribution may be obtained by solving the appropriate set of linear equations. The remainder of this Chapter will be devoted to examining solutions obtained by this method.

Analysis of Specific Network Types

At this point it is useful to introduce a schematic notation for representing queueing networks. Let empty circles denote service facilities having exponential service times, let rectangles denote the location of queues, and let the flow of customers through the network be indicated by arrows. Thus Figure 3-1 is intended to represent a network made up of two queues in series.

Figure 3-1

Two Queues in Series

The arrow entering the network represents customers arriving according to some stationary Poisson arrival process and the

arrow leaving the network at the right represents departing customers. Unless otherwise specified it will be assumed that the means of the various arrival and service processes are arbitrary and that there are no bounds on the maximum queue lengths. This particular network, consisting of only two queues, was examined in an early paper by O'Brien (63).

Expanding the notation, let a circle containing the letter P denote a service facility made up of an arbitrary number of identical exponential servers operating in parallel. That is, if there are p exponential servers in parallel each having mean rate u, then the service facility provides service that is exponential with mean rate $u \cdot \min(p,k)$ where k is the number of customers present at the facility. Figure 3-2 provides an example of this notation. As in Figure 3-1 no assumption is made about the relative number or rate of the servers at each facility. Networks of this type with an arbitrary number of queues were studied by R.R.P. Jackson (49).

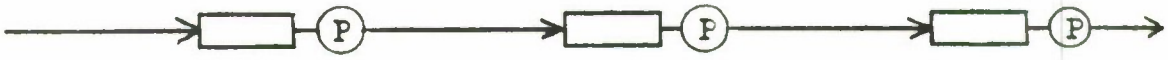


Figure 3-2

Parallel Servers

Continuing to expand the notation, let a circle with more than one arrow leaving it indicate customers may leave that service facility by taking any one of a number of paths.

For example, in Figure 3-3 a customer leaving service facility 3 can return to service facilities 1, 2 or 3, or can leave the network entirely.

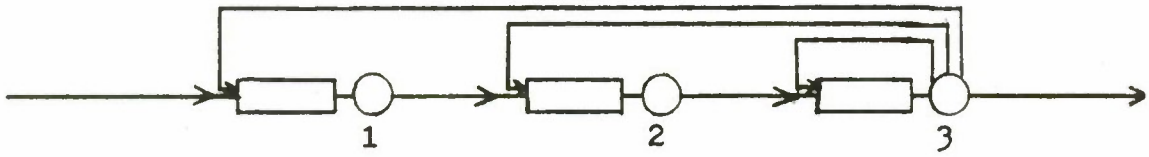


Figure 3-3

Terminal Feedback

Whenever such branch points appear in a network diagram it will be assumed that each particular path has associated with it a fixed probability and that, whenever a customer leaves the service facility, he selects the next path according to these probabilities independently of the choices that he or other customers may have made in the past. Naturally the sum of the probabilities associated with the different paths leaving a single service facility must be equal to one.

Figure 3-4 provides another example of multiple paths leaving a service facility. For notational simplicity only a single line is shown leaving each facility and the branching into separate paths appears further on.

Figure 3-4

Internal Feedback

Using these notational conventions the general network model of J.R. Jackson (47) is illustrated in Figure 3-5. There are an arbitrary number of service facilities (three are shown), each facility has an arbitrary number of identical parallel servers, and a customer may proceed to any service facility in the network after completing service at any given facility. In addition, new customers may enter the system via any queue and may leave the system at the completion of service at any facility.

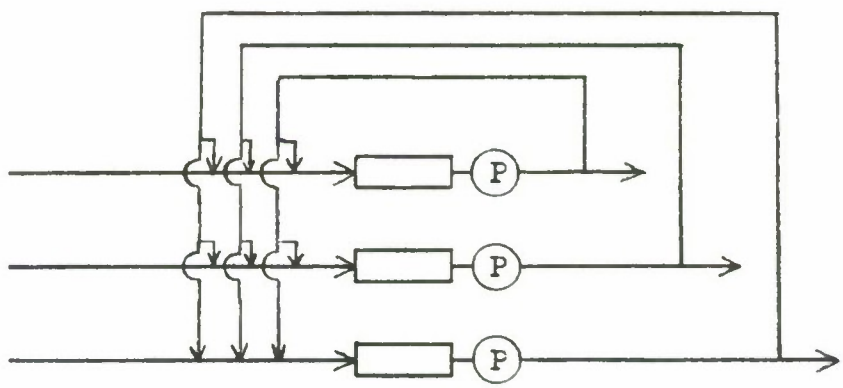


Figure 3-5

Arbitrarily Connected Queueing Network

Limitations on Network Capacity

In the work of O'Brien, R.R.P. Jackson and J.R. Jackson presented thus far, no limits are placed on the lengths of any of the queues appearing in the network diagrams. One method of imposing such limits is to construct a closed net-

work having the property that customers can neither enter nor leave. If such a network is then initialized with a given number of customers, these customers will circulate through the network indefinitely so that the number of customers in the network will not only be bounded but will in fact be constant.

Koenigsberg (58) introduced a closed system of a particularly simple form which he termed a cyclic queue. Koenigsberg's model is represented in Figure 3-6 for the case in which there are four service facilities. Actually Koenigsberg solved this model for an arbitrary number of service facilities and an arbitrary number of circulating customers. It is assumed in Figure 3-6 that each queue in the network has capacity equal to the total number of circulating customers.

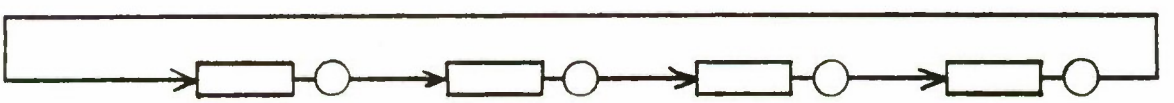


Figure 3-6

Cyclic Queue

Finch (37) adopted a slightly different approach by considering situations in which the total number of customers is bounded by some value but not restricted to always remain equal to that value. Finch analyzed both the terminal feed-

back networks of Figure 3-3 and the internal feedback networks of Figure 3-4. In both cases Finch assumed that the arrival process shut down whenever the total number of customers in the network reached some upper bound N. That is, rather than assuming a stationary arrival process Finch assumed the arrival process was a function of K, the total number of customers in the network. For $K < N$ the arrival process was assumed to be Poisson with constant mean rate, and for $K \geq N$ the arrival process was assumed to be Poisson with mean rate zero (i.e., no customers arrive).

GENERAL NETWORK MODELS

The Work of J.R. Jackson

In an impressively comprehensive paper J.R. Jackson (48) combined his earlier work with some ideas from Finch (37) and then went on to develop a solution technique for an extremely wide class of queueing networks. Jackson begins by considering the totally general model of inter-connecting paths illustrated in Figure 3-5. To this he adds Finch's notion of allowing the mean arrival rate to be a function of K, the total number of customers in the system. Working under the assumption of a non-stationary Poisson arrival process, Jackson then directs his attention to the case where the mean arrival rate is an arbitrary function of K. In contrast, Finch considers only the case where the mean arrival rate is given by a simple step function equal to u if $K < N$ and equal to 0 if $K \geq N$.

In the networks considered by Finch customers may only arrive at a single point as illustrated in Figures 3-3 and 3-4. Jackson on the other hand must contend with the possibility of customers arriving at any point as illustrated in Figure 3-5. To deal with this situation Jackson first assumes that the total arrival rate for the system is some arbitrary function $\lambda(K)$. He then assumes that arriving customers enter the system at service facility n with fixed probability $r(0,n)$ ($\sum_{n=1}^{N} r(0,n) = 1$ where N is the number

of service facilities in the system). Thus the mean arrival rate to the $n^{\underline{th}}$ service facility in Jackson's model is $\lambda(K) \cdot r(0,n)$.

Jackson also generalizes the notion of parallel servers at a service facility to include arbitrary exponential service. That is, when k customers are present at service facility n Jackson assumes only that the service time is exponentially distributed with mean rate u(n,k). As previously pointed out, if the $n^{\underline{th}}$ service facility consists of p exponential servers in parallel each having mean rate u_n, and if there are k customers present at the facility, then the service time is distributed exponentially with mean rate $u_n \cdot \min(p,k)$. Jackson's more general approach is to simply specify the mean rate by some arbitrary function u(n,k).

In addition to this synthesis and generalization of earlier work Jackson also introduces new mechanisms into his model which allow him to include closed networks such as those considered by Koenigsberg as a special case. The first mechanism is termed triggered arrivals. In a system with triggered arrivals it is assumed that there exists a positive integer K* which serves as a lower bound on the total number of customers present. That is, whenever the total number of customers in the system is equal to K* and a customer exits, a new customer is immediately injected into the system. The probability that this new customer

will arrive at service facility n is equal to r(0,n) for n=1,2,...,N . Clearly, if a triggered arrival system is constructed so that $\lambda(K) = 0$ if and only if $K \geq K^*$ then the resulting system can be interpreted as a closed queueing network with K* circulating customers. Koenigsberg's model thus becomes a highly specialized example of such a system.

Another mechanism which Jackson treats in this paper is known as service deletion. Under this mechanism it is assumed that associated with each service facility there is a positive integer k_n^* which acts as an upper bound on the number of customers that can be present at the facility. If the number of customers present at service facility n is equal to k_n^* and a new cutomer arrives, the customer currently being served is immediately ejected and then proceeds to his next destination according to the same set of probabilities that govern normal departures. Since service times are exponentially distributed and thus "memoryless", it is equivalent to assume in this case that the arriving customer, rather than the customer being served, is the one that is ejected from the service facility. Jackson thus provides one possible mechanism for limiting the size of queues in a network.

It should be pointed out that Jackson does not present explicit closed form solutions for the queueing networks he considers. Instead he presents a solution technique for solving the large set of homogeneous linear equations asso-

ciated with the Markov process which characterizes the networks. The solution technique, which is is more fully explained in Appendix B, involves solving a considerably smaller set of linear equations and then constructing the solution to the larger set of equations in a certain well defined manner. This greatly reduces the effort involved in solving specific problems and, in addition, demonstrates the existence of a number of structural properties which all solutions must possess. However Jackson's work does not entirely supersede the explicit closed form solutions obtained by Koenigsberg, Finch and others.

The Work of W.J. Gordon and G.F. Newell

The work of Gordon and Newell (41) illustrates how a particular subcase of Jackson's work may be profitably explored. Gordon and Newell consider queueing networks with completely general inter-connecting paths as illustrated in Figure 3-5, except that only closed systems are examined so there is no possibility of customers either arriving or departing.

In a sense this model represents a natural generalization of closed cyclic networks considered by Koenigsberg, whereas Jackson's model (48) represents a generalization of the open network model (47) he considered earlier. The mechanism of triggered arrivals then allows Jackson to include closed networks such as those of Koenigsberg as a

special case. Despite its elegant generality this treatment is somewhat cumbersome, and so in practice it is far simpler to follow the notation and equations of Gordon and Newell when solving problems which are initially defined in terms of closed networks. The solution technique presented in Appendix B closely parallels the derivation which Gordon and Newell present.

After re-deriving Jackson's equations* in a specialized form Gordon and Newell go on to explore the asymptotic behavior of closed systems as the number of circulating customers becomes very large. This aspect of their work is entirely new and lends further interest to the paper.

In a second paper published the same year Gordon and Newell (42) consider a network of cyclic queues of the type illustrated in Figure 3-6 with the additional restriction that the maximum queue length which can build up at a service facility is less than the total number of circulating customers. When a queue reaches its maximum permissible length, it is assumed that the service facility which feeds into that queue becomes blocked or in effect shuts down. In contrast the service deletion mechanism of Jackson (48) would imply in this case that, when a queue reached its maximum length, customers departing from the service facility which feeds into that queue would simply bypass it and proceed directly

*Gordon and Newell were unaware of the earlier work of Jackson (48) as indicated in (43).

to the next service facility in the cycle.

The induced blocking mechanism considered by Gordon and Newell is thus quite different from the service deletion mechanism of Jackson. Unfortunately the equilibrium equations for this problem turn out to be rather complex, and as a result explicit closed form solutions are obtained only for the case of a cyclic network of two queues. In addition the limiting behavior of such systems when the number of customers is small (i.e., when there is a low probability of a queue reaching its maximum length) and when the number of customers is large (i.e., when there is a high probability of a queue reaching its maximum length) is explored.

The problem of limited size queues with induced blocking also proved difficult to treat in the case of queues in series as illustrated in Figure 3-1. Hunt (46) considers such a system but is able to derive solutions only for the case of two queues with the second having arbitrary finite capacity, and the case of three queues with the second and third having a capacity of one. The rather limited success of these efforts suggests that this is a difficult problem to treat in general. However the close parallel between the induced blocking mechanism and the behavior of computer systems when memory becomes saturated should provide strong motivation for additional research in this area.

CHAPTER 4: INTRODUCTION TO THE CENTRAL SERVER MODEL

SPECIFICATION OF THE MODEL

Individual Program Behavior

One method of constructing a mathematical model of a complex physical system is to first analyze a particular component of the system in relative isolation and to then gradually introduce additional detail. In the case of multiprogramming systems it is convenient to begin this process by analyzing the behavior of an individual program running in a slightly simplified multiprogramming environment. It will be assumed that programs enter this environment by being loaded into main memory from a device such as a disk or card reader. Once loaded, a program presents the CPU with a sequence of instructions to be executed. Scattered through this sequence are a number of I/O transfer requests which, when encountered, cause the CPU to suspend instruction processing for the duration of the transfer. After a transfer has been completed a new interval of CPU processing begins, then another interval of I/O processing, and so on until the CPU eventually encounters a symbolic STOP statement. This terminates the program and causes it to exit from the system.

Now consider the effect of a multiprogramming environment on the preceding description of individual program

behavior. Since all multiprogramming environments are designed to maintain a number of programs in the active state at all times, it is possible for a program in such an environment to request service from a processor at a time when that processor is already busy serving some other program. Such overlapping requests for service will cause queues to build up from time to time at the various processors in the system.

Taking this additional consideration into account, program behavior is thus characterized by a period of initial loading followed by alternating intervals of CPU processing and I/O processing with each processing interval possibly preceded by a queueing delay, and eventually a final period of CPU processing after which the program exits from the system. This general behavior pattern is represented schematically in Figure 4-1 for a system in which programs are initially loaded from a card reader and may then carry out I/O processing on a disk, magnetic tape and data cell. Note that the behavior of any program in such a system can be described by a continuous path through this diagram beginning at the card reader and ending at the exit arrow. The exact structure of the program behavior path, as well as the amount of time required to service each processing request, will vary from program to program and is left unspecified. The time a program spends waiting in queues depends not on the program itself but on the

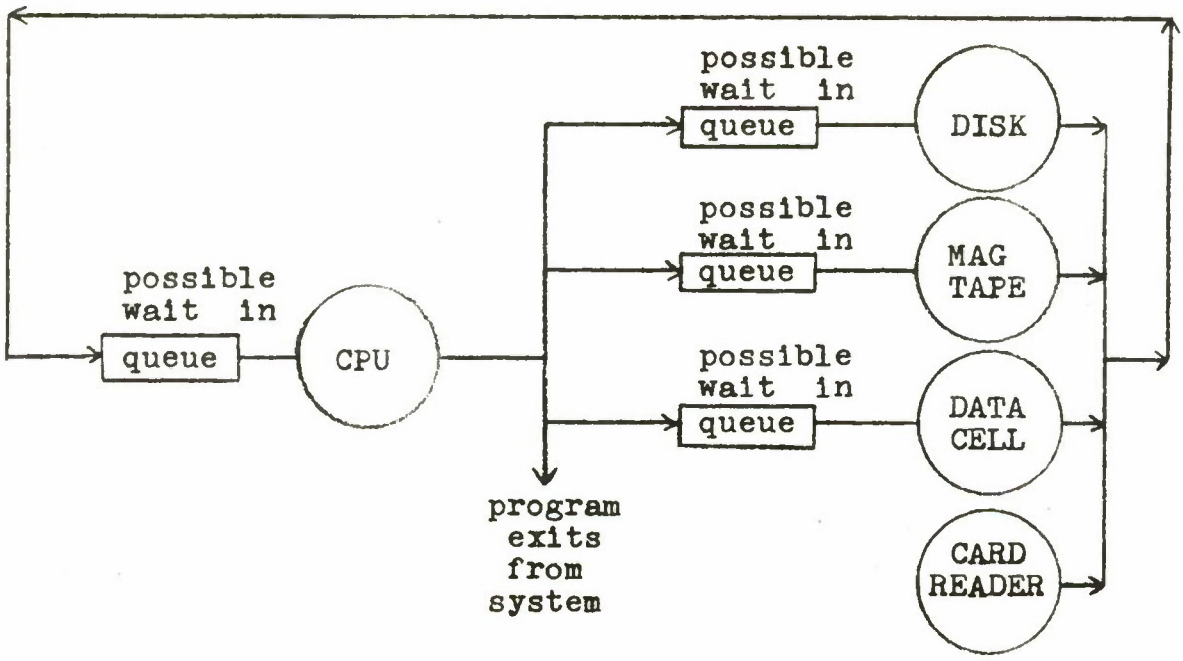


Figure 4-1 Program Behavior

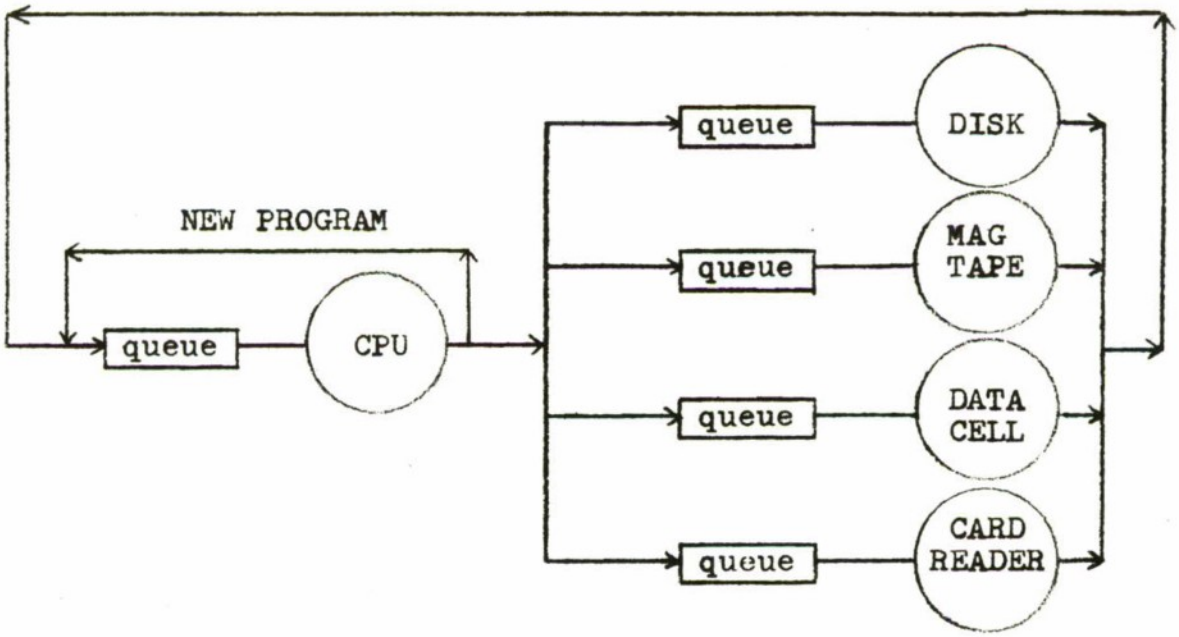


Figure 4-2 Memory Partition Behavior

activity of the other programs in the system. This too is left unspecified.

The circles labeled DISK, MAG TAPE and DATA CELL in Figure 4-1 are not intended to represent individual devices. Instead they represent individual peripheral processors, each capable of controlling a number of physical devices but each having the property of being able to carry out only one I/O transfer at a time. Thus a peripheral processor might correspond to a data channel combined with a device controller which is connected to several disk or magnetic tape drives. Note that even though each program normally has its own set of dedicated tape drives, it is presumed in Figure 4-1 that there is only one data channel/controller for all the tape drives in the system. Hence the magnetic tape processor is depicted as a shared resource subject to queueing delays in the same way that the disk and data cell are.

System Behavior

The model of program behavior represented in Figure 4-1 can be converted to a model of system behavior for an entire class of multiprogramming systems by making a few relatively minor alterations. First of all, rather than considering the processing requests generated by a particular program it is instead necessary to consider the processing requests generated by a particular memory partition in a multiprogramming system. This change affects the way in which program

terminations are represented. When a program encounters a symbolic STOP statement it simply exits from the system as indicated in Figure 4-1. However, the partition in which the program resided does not exit but is instead loaded with the next program awaiting execution. Thus, from the point of view of the memory partition, the termination of one program is followed by the loading of another.

In order to represent this phenomenon let the activity of a particular memory partition be described by a marker moving about the diagram in Figure 4-2. The location of the marker will correspond to the state of the program in the associated partition: either waiting in a queue or receiving service from a processor. When the CPU encounters a symbolic STOP statement and the program terminates, assume that the marker moves out along the NEW PROGRAM path leaving the CPU. The marker will then immediately return to the CPU queue, this time representing the first CPU processing request of the next program. Actually the first few processing requests the marker generates at this point will not correspond to the next program itself but rather to the processing activity required to load the next program into the partition. However, this processing activity will be regarded as a part of the next program for purposes of this discussion.

The model can now be extended to include the behavior of an entire multiprogramming system simply by assuming

that each memory partition in the system is represented by a different marker moving about the diagram in Figure 4-2. Note that only one marker can occupy a processor at any time, but any number of markers can occupy the queue associated with a given processor. Under these conventions the movement of markers along the paths of Figure 4-2 can be seen to correspond in a natural way to the operation of an actual multiprogramming system.

Multiprogramming systems which can be represented in this manner must satisfy certain restrictions. First of all, the number of main memory partitions in the system being modeled (i.e., the degree of multiprogramming) must be constant since there is no way of either adding or deleting markers in Figure 4-2. In addition, an individual program cannot undergo concurrent processing on more than one processor at a time because only one marker is associated with each program. Finally, the system must be operating under conditions of full load since it is assumed that there is always a new program ready to begin processing when a currently active program terminates. Note that these restrictions do not interfere with the primary objective of the model which is to represent cases in which a number of active programs are present in a single system at the same time. Since this state of affairs is usually regarded as the most significant aspect of any multiprogramming environment, the three restrictions just cited do not

prevent the model from being of both practical and theoretical interest.

Behavior Parameters

Figure 4-2 cannot be regarded as a complete description of a multiprogramming model because it does not specify the nature of the paths that markers follow as they move about the diagram or the amount of time that markers spend at each processor they encounter. These two factors correspond to the sequence of I/O processing requests generated by a program and the amount of time necessary to service individual CPU and I/O processing requests.

There are a number of ways in which the sequence of I/O requests generated by a program can be specified. For example, it is possible to observe an actual system over a period of time, note in detail the path followed by each program, and then include all this information in the model. This representation would be entirely accurate but would result in an unwieldy model since a vast number of parameters would be necessary to permit this information to be encoded. It is thus essential to develop a more concise representation of program behavior, even if this entails some sacrifice in the fidelity of representation of the final model.

To see how this might be done imagine that an actual system is observed for some period of time, but assume that

the only information collected is the relative frequencies with which various paths are selected by programs completing CPU processing requests. A typical set of such data for the system represented in Figure 4-2 is presented in Table 4-1.

Path Designation	Relative Frequency
NEW PROGRAM path	1 in 20
Path to DISK	9 in 20
Path to MAG TAPE	5 in 20
Path to DATA CELL	2 in 20
Path to CARD READER	3 in 20

Table 4-1

Relative Frequency with which Programs Completing CPU Processing Requests Select Various Paths

To a certain extent the program behavior paths which generated this data can be reconstructed by assuming in Figure 4-2 that whenever a marker leaves the CPU its next path is determined by probabilities which correspond to the relative frequencies in Table 4-1. That is, assume that each time a marker leaves the CPU the probability of its selecting the NEW PROGRAM path is 1/20, the probability of its selecting the path to DISK is 9/20, and so on. A system

operating in this manner clearly will, in the long run, utilize its peripheral processors with the same relative frequency as the observed system. In addition, individual program behavior will exhibit considerable variability since the total number of processing requests per program and the sequence of processing requests within each program are both determined by random factors. Thus, even though the exact details of specific program behavior are lost under these assumptions, a substantial link with reality is maintained. Furthermore the number of parameters necessary to specify program behavior is significantly reduced since only the path selection probabilities need be supplied.

Specifying the amount of time a marker spends at each processor is a more difficult problem. In actual multiprogramming systems the amount of service time per processing request is likely to be a rather complex function which differs from one processor to another. It is of course possible to observe an actual system and empirically obtain the distribution of service times for each processor. These empirical distributions could then be approximated by continuous functions and the service time for each processor could then be specified as a random variable having the associated continuous distribution function.

There are two major drawbacks to this approach. First of all, analysis of queueing networks of the type illustrated in Figure 4-2 for arbitrarily distributed service times is

almost certain to lead to a mathematically intractable situation. Second, and in some sense equally as important, a set of continuous distribution functions fitted to some arbitrary body of data would be an extremely awkward collection of parameters to incorporate into a model, especially if the effects of parameter variation were being explored with the aim of generalizing to other systems. Thus, as in the case of program behavior paths, it is again necessary to seek a more concise and mathematically tractable representation, even at the expense of sacrificing fidelity of representation in the final model.

There are two features of the actual service time distributions which seem especially critical and which will be incorporated in the model. The first is that each processing request directed to a given processor is likely to require a different amount of time for its completion. That is, the service times associated with a given processor are not all identical but instead vary from one request to another. When these variations are averaged together, a second important feature becomes apparent which is that the average amount of service time per processing request is not necessarily the same for each processor in the system.

Assume that the average amount of service time per processing request is $1/u_o$ for the CPU and $1/u_j$ for the $j^{\underline{th}}$ PPU. From a mathematical standpoint the simplest way to incorporate these parameters into the model, while also including

variability in the individual service times of each processor, is to assume that the service time per processing request for the CPU is an exponentially distributed random variable* with mean $1/u_o$ and that the service time per processing request for the j^{th} PPU is an exponentially distributed random variable with mean $1/u_j$. Note that it is not being asserted that these exponential distributions provide an entirely accurate representation of the actual service time functions, but only that they include the most significant aspects of these functions. The point is that the model should not be judged on the goodness-of-fit of the exponential assumptions, but rather on the validity and utility of the insights which are ultimately derived.

Summary Description of the Model

The final model, which is represented schematically in Figure 4-3, may be described in the conventional terminology of queueing theory as a closed queueing network of L+1 exponential servers and N circulating customers. Customers leaving the o^{th} (central) server proceed to the j^{th} server with probability p_j $(j=0,1,\ldots,L)$, and customers leaving one of the L peripheral servers proceed directly to the central server with probability one. The parameters of the

*Appendix A contains a detailed discussion of the most significant features of exponentially distributed random variables.

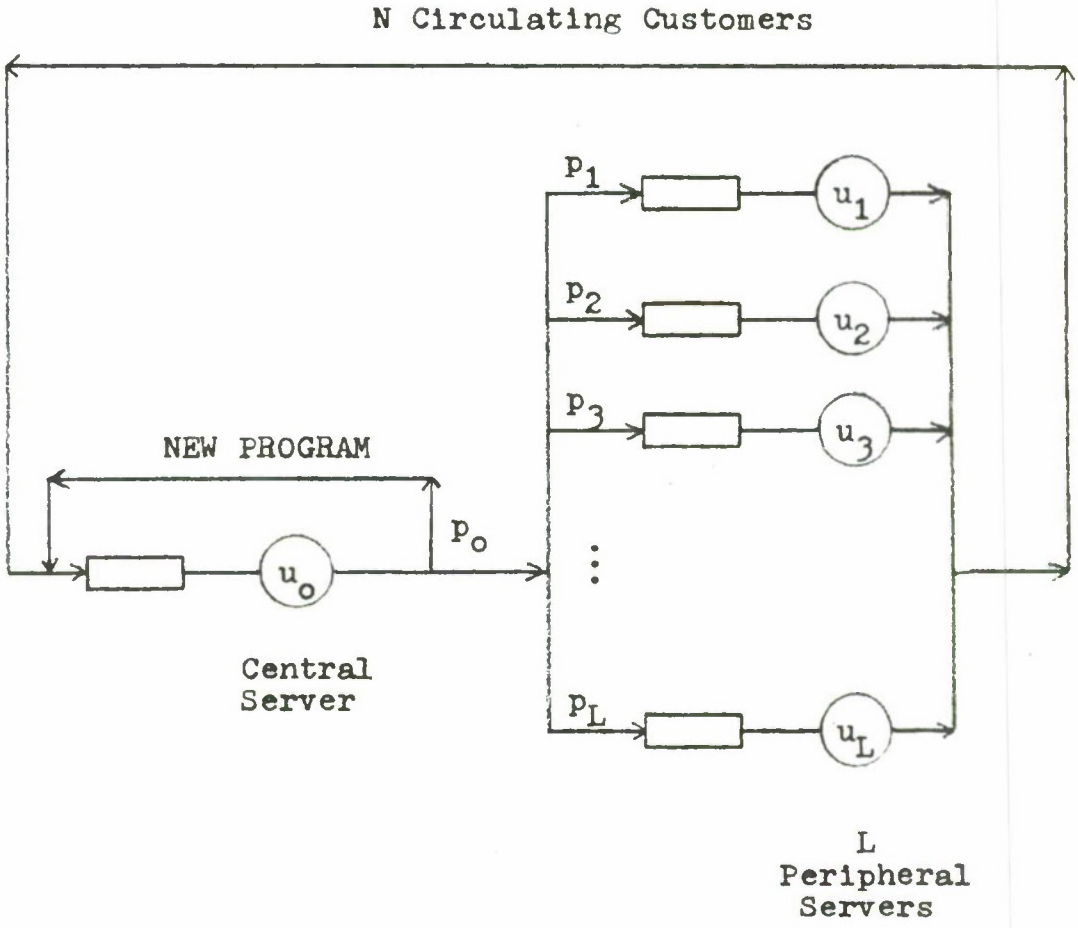


Figure 4-3
Central Server Model of Multiprogramming

system are:

N - number of circulating customers (i.e., degree of multiprogramming)

L - number of peripheral servers

p_j - probability that a customer proceeds to the $j^{\underline{th}}$ server after leaving the central server (j=0,1, ...,L)

u_j - mean rate of the $j^{\underline{th}}$ server (j=0,1,...,L)
$1/u_j$ is the average time required to complete a service request on the $j^{\underline{th}}$ server, and the probability that a service request on the $j^{\underline{th}}$ server has length $\leq$ T is

$$\int_0^T u_i \, e^{-u_i t} \, dt$$

For purposes of this discussion the class of queueing networks which satisfy the preceding description will be known as central server networks and the models based on these networks will be known as central server models. The $o^{\underline{th}}$ (central) server in a central server network will be referred to as the CPU and the $j^{\underline{th}}$ (peripheral) server in a central server network will be referred to as the $j^{\underline{th}}$ PPU.

ELEMENTARY PROPERTIES

Introduction

The primary reason for developing the central server model of multiprogramming is to gain some understanding of the nature of the queueing delays which arise in multiprogramming systems. Before analyzing this aspect of the model certain other properties will be considered which are in a sense more elementary in that they are independent of queueing delays and can be derived without considering the steady state distribution of customers in the network. These properties, which have to do with the distribution of the number of processing requests per program and the total processing time per program, will help to further introduce the model and will also be of some practical use later in the analysis. Table 4-2 (p. 98) provides a convenient summary of the main results of this section.

Distribution of Processing Requests

To begin the analysis recall that each time a program completes a CPU processing request the probability that the NEW PROGRAM path will be selected is p_o. Since selection of the NEW PROGRAM path corresponds to termination of a program, it is relatively simple to obtain the distribution of CPU processing requests per program.

Suppose a program has just been loaded into main memory. The probability that the program will make exactly one request for CPU processing is the probability that it terminates (i.e., selects the NEW PROGRAM path) immediately after completing its first CPU processing request. According to the model the probability of this happening is p_o. Similarly, a program making exactly two requests for CPU processing must select a path other than the NEW PROGRAM path after its first CPU processing request - an event which occurs with probability $1-p_o$ - and then select the NEW PROGRAM path after its second CPU processing request. Thus the probability of a program making exactly two CPU processing requests is $(1-p_o)p_o$.

In general a program which makes exactly n requests for CPU processing must select a path other than the NEW PROGRAM path n-1 consecutive times and then select the NEW PROGRAM path the $n^{\underline{th}}$ time. Since each path selection decision is independent of all other decisions, the probability that this event will occur is $(1-p_o)^{n-1}p_o$, and thus the expected number of CPU processing requests per program is

$$\sum_{n=1}^{\infty} n\, p_o\, (1-p_o)^{n-1} = 1/p_o \qquad 4\text{-}1$$

Obtaining the number of PPU processing requests per program is a bit more complicated. Consider the probability that a program makes exactly n requests for processing from the $j^{\underline{th}}$ PPU, and suppose for the moment that the total number

of CPU processing requests made by the program is C+1. That is, after each of the first C requests for CPU service the program selects a path to a PPU - possibly the $j^{\underline{th}}$ one - and after the $C+1^{\underline{st}}$ request the program terminates by selecting the NEW PROGRAM path. Note that if this program is to make a total of n requests for service from the $j^{\underline{th}}$ PPU it is necessary to have $C \geq n$.

Consider the first C requests for CPU processing. After n of them the program must select the $j^{\underline{th}}$ PPU, and after C-n of them the program must select a PPU other than the $j^{\underline{th}}$. Since the NEW PROGRAM path cannot be selected during this period, the probability of selecting a PPU other than the $j^{\underline{th}}$ is $(1-p_o-p_j)$ and the probability of selecting the $j^{\underline{th}}$ PPU remains p_j. It then follows that the probability of making n consecutive requests for service from the $j^{\underline{th}}$ PPU followed by C-n consecutive requests for service from a PPU other than the $j^{\underline{th}}$ is $p_j^{\,n}(1-p_o-p_j)^{C-n}$.

Of course there is no reason in this case to require that the n requests for service from the $j^{\underline{th}}$ PPU precede the C-n other requests, and in fact any ordering of these C requests is legitimate as long as exactly n requests for service from the $j^{\underline{th}}$ PPU occur. Since the total number of such orderings is $\binom{C}{n}$ and the probability of obtaining any particular ordering is $p_j^{\,n}(1-p_o-p_j)^{C-n}$, the probability that there will be n requests for the $j^{\underline{th}}$ PPU somewhere among the first C requests for PPU processing is

$\binom{C}{n} p_j^{\,n}(1-p_o-p_j)^{C-n}$. Since it is also being assumed that the program makes exactly C+1 requests for CPU service, it is necessary to multiply this expression by p_o which is the probability that the NEW PROGRAM path will be selected after the next (i.e., the C+1st) CPU processing request. Thus the probability that a program makes C+1 requests for CPU service and n requests for service from the jth PPU is $p_o\binom{C}{n} p_j^{\,n}(1-p_o-p_j)^{C-n}$.

To obtain the probability that a program makes exactly n requests for service from the jth PPU irrespective of the number of requests it makes for CPU service, it is necessary to sum the preceding expression over all values of C for which n requests for service from the jth PPU are possible (i.e., for C≥n). Thus the probability a program makes n requests for service from the jth PPU is*

$$\sum_{C=n}^{\infty} \binom{C}{n} p_j^{\,n}(1-p_o-p_j)^{C-n} p_o = \frac{p_o}{p_o+p_j}\left(\frac{p_j}{p_o+p_j}\right)^n \qquad 4\text{-}2$$

---

*Equation 4-2 follows from the observation that

$$\sum_{C=n}^{\infty}\binom{C}{n} x^{C-n} = 1/(1-x)^{n+1}$$

for 0<x<1. This result can be derived by differentiating both sides of the equation

$$\sum_{C=0}^{\infty} x^C = 1/(1-x)$$

n times with respect to x and then dividing through by n!

From this expression it is possible to obtain the expected number of requests for service from the $j^{\underline{th}}$ PPU as

$$\sum_{n=0}^{\infty} n \frac{p_o}{p_o+p_j}\left(\frac{p_j}{p_o+p_j}\right)^n = p_j/p_o \qquad 4\text{-}3$$

Note that the summation in equation 4-3 begins with n=0 since it is possible for a program to make no requests for service from the $j^{\underline{th}}$ PPU. On the other hand the summation in equation 4-1 begins with n=1 since each program must make at least one request for CPU processing.

One of the implications of equation 4-3 is that the total expected number of PPU processing requests per program is

$$\sum_{j=1}^{L} p_j/p_o = \frac{1}{p_o}\sum_{j=1}^{L} p_j = \frac{1}{p_o}(1-p_o) = \frac{1}{p_o} - 1 \qquad 4\text{-}4$$

This makes sense intuitively since the total expected number of CPU processing requests per program is $1/p_o$ and since, in any program, the number of PPU processing requests is one less than the number of CPU processing requests.

Distribution of Total Processing Time

Now that the distribution of the number of processing requests per program has been obtained for each server it is possible to obtain the distribution of total processing time per program. Again it is simpler to begin with the CPU. Clearly the probability that the total CPU processing time per program is less than or equal to t is the probability

that a program requires exactly one CPU processing interval times the probability that this interval is less than or equal to t plus the probability that a program requires exactly two CPU processing intervals times the probability that the sum of these two intervals is less than or equal to t and so on. If $D_n(t)$ is the probability that the sum of n CPU processing intervals is less than or equal to t and $D_T(t)$ is the probability that the total CPU processing time per program is less than or equal to t, then the preceding sentence may be expressed more concisely as

$$D_T(t) = \sum_{n=1}^{\infty} p_o(1-p_o)^{n-1} D_n(t) \qquad 4\text{-}5$$

Taking the Laplace-Stieltjes transform of both sides,

$$L_{D_T}(s) = \sum_{n=1}^{\infty} p_o(1-p_o)^{n-1} L_{D_n}(t)$$

Since the CPU processing intervals are independent and identically distributed random variables, it follows from one of the elementary properties of Laplace-Stieltjes transforms that

$$L_{D_n}(s) = (L_{D_1}(s))^n$$

Hence

$$L_{D_T}(s) = \sum_{n=1}^{\infty} p_o(1-p_o)^{n-1} (L_{D_1}(s))^n$$

$$= \frac{p_o \, L_{D_1}(s)}{1 - (1-p_o) \, L_{D_1}(s)}$$

Since CPU processing intervals are exponentially distributed with mean $1/u_o$

$$D_1(t) = \int_o^t u_o\, e^{-u_o x}\, dx$$

Thus $L_{D_1}(s) = \int_o^\infty e^{-st}\, dD_1(t)$

$$= \int_o^\infty e^{-st}\, u_o\, e^{-u_o t}\, dt = \frac{u_o}{u_o+s}$$

Substituting,

$$L_{D_T}(s) = \frac{p_o \frac{u_o}{u_o+s}}{1 - (1-p_o)\frac{u_o}{u_o+s}} = \frac{u_o p_o}{u_o p_o + s} \qquad 4\text{-}6$$

As can be readily verified, the inversion of equation 4-6 yields

$$D_T(t) = \int_o^t u_o p_o\, e^{-u_o p_o x}\, dx$$

That is, the total amount of CPU processing time per program is an exponentially distributed random with mean $\frac{1}{u_o p_o}$.

A similar argument can be used to determine the distribution of the total amount of processing time per program on the $j^{\underline{th}}$ PPU. The major difference is that it is now possible - with probability $p_o/(p_o+p_j)$ - for a program to make no requests for processing on the $j^{\underline{th}}$ PPU during the course of its execution. Hence the formula corresponding to equation 4-5 for the $j^{\underline{th}}$ PPU is

$$D_T(t) = \frac{p_o}{p_o + p_j} + \sum_{n=1}^{\infty} \frac{p_o}{p_o+p_j}\left(\frac{p_j}{p_o+p_j}\right)^n D_n(t) \qquad 4\text{-}7$$

Note that when equation 4-7 is evaluated at t=0 the resulting probability value is $p_o/(p_o+p_j)$. This is the probability that the total processing time per program on the $j^{\underline{th}}$ PPU is equal to zero. Hence the probability that the total processing time is less than or equal to t always includes the term $p_o/(p_o+p_j)$.

Taking the Laplace-Stieltjes transform of equation 4-7

$$L_{D_T}(s) = \frac{p_o}{p_o+p_j} + \sum_{n=1}^{\infty} \frac{p_o}{p_o+p_j}\left[\frac{p_j}{p_o+p_j}\right]^n L_{D_n}(s)$$

$$= \sum_{n=0}^{\infty} \frac{p_o}{p_o+p_j}\left[\frac{p_j}{p_o+p_j} L_{D_1}(s)\right]^n$$

$$= \frac{\frac{p_o}{p_o+p_j}}{1 - \frac{p_j}{p_o+p_j} L_{D_1}(s)}$$

Since service intervals on the $j^{\underline{th}}$ PPU are exponentially distributed with mean $1/u_j$, it follows that

$$D_1(t) = \int_o^t u_j\, e^{-u_j x}\, dx$$

and $$L_{D_1}(s) = \frac{u_j}{u_j+s}$$

Thus $$L_{D_T}(s) = \frac{\frac{p_o}{p_o+p_j}}{1 - \frac{p_j}{p_o+p_j}\frac{u_j}{u_j+s}}$$

$$= \frac{u_j p_o + p_o s}{u_j p_o + p_o s + p_j s}$$

$$= \frac{p_j}{p_o+p_j}\left[\frac{\frac{u_j p_o}{p_o+p_j}}{\frac{p_o u_j}{p_o+p_j} + s}\right] + \frac{p_o}{p_o+p_j}$$

Inverting,

$$D_T(t) = \frac{p_j}{p_o+p_j}\int_o^t \frac{u_j p_o}{p_o+p_j} e^{\frac{-u_j p_o}{p_o+p_j} x} dx + \frac{p_o}{p_o+p_j} \qquad 4\text{-}8$$

Put into words, equation 4-8 expresses the fact that the total processing time per program on the $j^{\underline{th}}$ PPU is zero with probability $p_o/(p_o+p_j)$ and, with probability $p_j/(p_o+p_j)$, is exponentially distributed with mean $(p_o+p_j)/(u_j p_o)$. Thus the expected processing time per program on the $j^{\underline{th}}$ PPU is

$$\frac{p_j}{p_o+p_j}\frac{p_o+p_j}{u_j p_o} + \frac{p_o}{p_o+p_j}\cdot 0 = \frac{p_j}{u_j p_o}$$

The results derived in this section are summarized in Table 4-2. There is a sense in which these results should not be considered as consequences of the central server model but rather as additional assumptions about program behavior which are implicit in the model. To elaborate upon this point, recall that the underlying purpose behind the central server model is to gain insight into the nature of the queueing delays which arise in multiprogramming systems.

	For the CPU	For the j^{th} PPU
Probability of exactly n service requests in a program	$p_o(1-p_o)^{n-1}$	$\frac{p_o}{p_o+p_j}\left(\frac{p_j}{p_o+p_j}\right)^n$
Expected number of service requests in a program	$\frac{1}{p_o}$	$\frac{p_j}{p_o}$
Probability that total service time per program is $\leq T$	$\int_0^T u_o p_o\, e^{-u_o p_o t}\, dt$	$\frac{p_j}{p_o+p_j}\int_0^T \frac{u_j p_o}{p_o+p_j}\, e^{\frac{-u_j p_o}{p_o+p_j} t}\, dt + \frac{p_o}{p_o+p_j}$
Expected total service time per program	$\frac{1}{u_o p_o}$	$\frac{p_j}{u_j p_o}$

Table 4-2

Derived Results Concerning Program Behavior

Instead of applying the model to this problem, the work of this section has been devoted to studying the behavior of programs operating in systems which satisfy the basic assumptions of the model.

Now, the basic assumptions of the model regarding program behavior were assumed to be sufficiently realistic to permit the model to be of value in exploring the effect of queueing delays on system performance. However, this does not imply that these assumptions are sufficiently realistic to permit the model to be of value in further exploring program behavior itself. Hence these derived results should not be interpreted as intrinsically useful information about program behavior in actual systems. Instead they should be regarded as additional constraints on program behavior which systems represented by the model must to some extent satisfy. This point is certainly not a major one, but it may help to clarify the relationship between this section and the remainder of the thesis.

CHAPTER 5: THE STEADY STATE DISTRIBUTION AND ITS PROPERTIES

ANALYTIC EXPRESSIONS

Derivation of the Steady State Distribution

Steady state distributions were discussed in general terms in Chapter 2. Chapter 3 then reviewed a number of specific queueing networks for which steady state distributions have been explicitly obtained. Central server networks were not included in this discussion since the literature contains no specific references to networks of this type. However, central server networks fall within the general class of queueing systems analyzed by Jackson (48) and Gordon and Newell (41), and hence it is possible to use the solution technique developed by these authors* to obtain the steady state distribution for this particular network type.

To apply the solution technique outlined in Appendix B to a specific queueing network it is first necessary to specify the matrix $P = (p_{ij})$ where p_{ij} is the probability that a customer leaving the i^{th} server will proceed to the j^{th} server. For central server networks the matrix P is defined as follows:

---

*This technique is reviewed in Appendix B.

$$P = \begin{bmatrix} p_o & p_1 & p_2 & \cdots & p_L \\ 1 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix} \qquad 5\text{-}1$$

The next step is to determine the solution of the equation $\underline{y} = \underline{y} \cdot P$ (i.e., equation B-7 of Appendix B). Given a matrix of the form specified in equation 5-1, it is easily verified that the vector $\underline{y} = (y_o, p_1 y_o, p_2 y_o, \ldots, p_L y_o)$ satisfies the equation $\underline{y} = \underline{y} \cdot P$ for any value of y_o. In particular the vector $\underline{y} = (u_o, p_1 u_o, p_2 u_o, \ldots, p_L u_o)$ satisfies this equation.

Next let $P(n_o, n_1, \ldots, n_L)$ denote the steady state probability that there are n_j customers at the $j^{\underline{th}}$ server in a central server network. It then follows immediately from equation B-8 that

$$P(n_o, n_1, \ldots, n_L) = \frac{1}{G(N)} \prod_{j=o}^{L} (y_j/u_j)^{n_j}$$

$$= \frac{1}{G(N)} \prod_{j=1}^{L} (p_j u_o/u_j)^{n_j} \qquad 5\text{-}2$$

where G(N) is a normalizing constant. Note that the multiplicative index j can begin at 1 since $y_o/u_o = u_o/u_o = 1$.

The normalizing constant G(N) is selected so that the sum of all the $P(n_o, n_1, \ldots, n_L)$ will be equal to one. Since

any value of $P(n_0,n_1,\ldots,n_L)$ for which $\sum_{j=0}^{L} n_j = N$ represents a possible state of the system, it follows that

$$G(N) = \sum_{\sum_{j=0}^{L} n_j = N} \prod_{j=1}^{L} (p_j u_o / u_j)^{n_j}$$

$$= \sum_{\sum_{j=1}^{L} n_j \leq N} \prod_{j=1}^{L} (p_j u_o / u_j)^{n_j} \qquad 5\text{-}3$$

Note that neither n_o nor p_o appears on the right hand sides of equations 5-2 and 5-3 although it is of course understood that $n_o = N - \sum_{j=1}^{L} n_j$ and $p_o = 1 - \sum_{j=1}^{L} p_j$. Equations 5-2 and 5-3 are also valid for central server networks in which the NEW PROGRAM loop is missing since such networks simply correspond to the case in which $p_o = 0$ and $\sum_{j=1}^{L} p_j = 1$.

Part of the value of central server networks is the extreme simplicity with which the solution to the equation $\underline{y} = \underline{y} \cdot P$ can be expressed. This simplicity makes it possible to analyze the steady state distribution in detail and to derive a number of related properties which are valid for all central server networks but which are not necessarily valid for the more general networks studied by Jackson and by Gordon and Newell. The next few sections

deal with some of these properties.

Processor Utilization

Let A_j denote the steady state probability that the j^{th} processor is active (i.e., not idle). Since the j^{th} processor is active if and only if $n_j \geq 1$

$$A_j = \sum_{n_j \geq 1} P(n_o, n_1, \ldots, n_L)$$

In particular,

$$A_o = \sum_{n_o \geq 1} P(n_o, n_1, \ldots, n_L)$$

Since $n_o \geq 1$ implies $\sum_{i=1}^{L} n_i \leq N-1$

$$A_o = \sum_{\sum_{i=1}^{L} n_i \leq N-1} P(n_o, n_1, \ldots, n_L)$$

$$= \sum_{\sum_{i=1}^{L} n_i \leq N-1} \frac{1}{G(N)} \prod_{i=1}^{L} (p_i u_o / u_i)^{n_i}$$

$$= \frac{G(N-1)}{G(N)} \qquad 5\text{-}4$$

For $j=1,2,\ldots,L$

$$A_j = \sum_{n_j \geq 1} P(n_o, n_1, \ldots, n_L)$$

$$= \sum_{\sum_{i=1}^{L} n_i \leq N \;\&\; n_j \geq 1} \frac{1}{G(N)} \prod_{i=1}^{L} (p_i u_o / u_i)^{n_i}$$

Factoring out the quantity $p_j u_o / u_j$ which appears in each term of the sum as a result of the fact that $n_j \geq 1$,

$$A_j = \frac{1}{G(N)} \frac{p_j u_o}{u_j} \sum_{\sum_{i=1}^{L} n_i \leq N-1} \prod_{i=1}^{L} (p_i u_o / u_i)^{n_i}$$

$$= \frac{p_j u_o}{u_j} \frac{G(N-1)}{G(N)}$$

$$= \frac{p_j u_o}{u_j} A_o$$

Thus

$$A_o u_o p_j = A_j u_j \qquad 5\text{-}5$$

Conservation Laws

Equation 5-5 has an interesting intuitive interpretation. Suppose that a central server network in equilibrium is observed for some interval of time of length T. Then the expected amount of time that the central server is

active during this interval is $A_o T$. Since the average amount of time required to process a customer through the central server is $1/u_o$, the expected number of customers processed by the central server during the interval is $A_o T/(1/u_o) = A_o T u_o$. Next note that each customer processed by the central server has probability p_j of being channeled to the $j^{\underline{th}}$ peripheral server. Thus the expected number of customers channeled to the $j^{\underline{th}}$ peripheral server during the interval is $A_o T u_o p_j$.

On the other hand the expected amount of time that the $j^{\underline{th}}$ peripheral server is active during the interval is $A_j T$, and so the expected number of customers processed by that server during the interval is $A_j T/(1/u_j) = A_j T u_j$. If T is large and the system is in equilibrium, then conservation of flow considerations would indicate that the number of customers channeled to the $j^{\underline{th}}$ peripheral server should equal the number of customers processed by that server. In other words $A_o T u_o p_j$ should equal $A_j T u_j$. This is equivalent to stating that $A_o u_o p_j = A_j u_j$. Since equation 5-5 may be obtained in this manner using conservation of flow considerations, this equation will be referred to as the Conservation Law for the remainder of this discussion.

In order to discuss one of the applications of the Conservation Law it is necessary to introduce the notion of the relative saturation of a server in a central server network. Essentially, relative saturation is the ratio between

the relative load on the server and the server's processing speed. If the relative load on the central server is defined as 1, then the relative load on the $j^{\underline{th}}$ peripheral server will be p_j since, over a long period of time, the ratio of the number of customers processed by the central server to the number of customers processed by the $j^{\underline{th}}$ peripheral server will approach $1/p_j$. Hence the relative saturation of the $j^{\underline{th}}$ peripheral server will be defined as p_j/u_j and the relative saturation of the central server will be defined as $1/u_o$.

Next note that the Conservation Law can be reformulated as follows:

$$\frac{A_o}{1/u_o} = \frac{A_1}{p_1/u_1} = \frac{A_2}{p_2/u_2} = \cdots = \frac{A_L}{p_L/u_L} \qquad 5\text{-}6$$

It is immediately obvious from equation 5-6 that the most highly saturated server is always the most highly utilized server and that equally saturated servers will be equally utilized. In fact utilization is directly proportional to relative saturation. It is also true that the most highly saturated - and highly utilized - server has the largest expected queue, but the proof of this fact will be deferred until later.

Before discussing the issue of expected queue lengths it is useful to introduce a powerful generalization of the Conservation Law. Begin by defining A_j^k as the steady state probability that there are k or more customers present at

the $j^{\underline{th}}$ server. Note that $A_j^o = 1$ and, according to the earlier definition of A_j, $A_j^1 = A_j$.

Now, $$A_o^k = \sum_{n_o \geq k} P(n_o, n_1, \ldots, n_L)$$

Since $n_o \geq k$ is equivalent to $\sum_{i=1}^{L} n_i \leq N-k$,

$$A_o^k = \sum_{\sum_{i=1}^{L} n_i \leq N-k} \frac{1}{G(N)} \prod_{i=1}^{L} (p_i u_o / u_i)^{n_i}$$

$$= \frac{G(N-k)}{G(N)} \qquad 5\text{-}7$$

For $j=1,2,\ldots,L$

$$A_j^k = \sum_{\sum_{i=1}^{L} n_i \leq N \;\&\; n_j \geq k} P(n_o, n_1, \ldots, n_L)$$

$$= \sum_{\sum_{i=1}^{L} n_i \leq N \;\&\; n_j \geq k} \frac{1}{G(N)} \prod_{i=1}^{L} (p_i u_o / u_i)^{n_i}$$

Factoring out the quantity $(p_j u_o / u_j)^k$ which appears in each term of the sum as a result of the fact that $n_j \geq k$,

$$A_j^k = \frac{1}{G(N)} \left[\frac{p_j u_o}{u_j}\right]^k \sum_{\sum_{i=1}^{L} n_i \leq N-k} \prod_{i=1}^{L} (p_i u_o / u_i)^{n_i}$$

$$= \left[\frac{p_j u_o}{u_j}\right]^k \frac{G(N-k)}{G(N)}$$

$$= \left[\frac{p_j u_o}{u_j}\right]^k A_o^k \qquad 5\text{-}8$$

Equation 5-8 may be thought of as a generalization of equation 5-5 since the latter can be derived from the former by setting k=1. For this reason equation 5-8 will be referred to as the Generalized Conservation Law. Note that once G(0),G(1),...,G(L) are known, equations 5-7 and 5-8 can be used to determine all the A_j^k. The A_j^k can then be used to determine the marginal distribution of customers at each server since the probability that there are exactly k customers at the $j^{\underline{th}}$ server is equal to $A_j^k - A_j^{k+1}$.

Queue Lengths

Define Q_j to be the expected number of customers present at the $j^{\underline{th}}$ server at equilibrium. Q_j may be interpreted as the expected length of the queue at the $j^{\underline{th}}$ server as long as queue length is understood to include the customer currently being served. Since $A_j^k - A_j^{k+1}$ is

the steady state probability that there are exactly k customers at the $j^{\underline{th}}$ server for k=0,1,...,N-1 and A_j^N is the steady state probability that there are exactly N customers at the $j^{\underline{th}}$ server, it follows that

$$Q_j = \sum_{k=0}^{N-1} k(A_j^k - A_j^{k+1}) + N \cdot A_j^N \qquad 5\text{-}9$$

$$= \sum_{k=0}^{N} k \cdot A_j^k - \sum_{k=1}^{N} (k-1) A_j^k$$

$$= \sum_{k=1}^{N} A_j^k$$

For j=0,

$$Q_o = \sum_{k=1}^{N} A_o^k$$

$$= \sum_{k=1}^{N} \frac{G(N-k)}{G(N)}$$

For j=1,2,...,L the Generalized Conservation Law implies

$$Q_j = \sum_{k=1}^{N} (p_j u_o / u_j)^k \, A_o^k$$

$$= \sum_{k=1}^{N} (p_j u_o / u_j)^k \, \frac{G(N-k)}{G(N)}$$

Hence, assuming G(0),G(1),...,G(N) have been determined, the expected queue length for each processor can be obtained

by evaluating the polynomial

$$q(x) = \sum_{k=1}^{N} x^k \frac{G(N-k)}{G(N)} \qquad 5\text{-}10$$

at the appropriate value of x. That is, $Q_o = q(1)$ and $Q_j = q(p_j u_o/u_j)$ for $j=1,2,\ldots,L$.

Since all the coefficients in the polynomial q(x) are positive,

$$\frac{p_j}{u_j} > \frac{p_k}{u_k} \iff q(p_j u_o/u_j) > q(p_k u_o/u_k) \iff Q_j > Q_k$$

Also $$\frac{1}{u_o} > \frac{p_j}{u_j} \iff 1 > p_j u_o/u_j$$

$$\iff q(1) > q(p_j u_o/u_j) \iff Q_o > Q_j$$

Thus the most highly saturated - and most highly utilized - server has the largest expected queue, and equally saturated servers have equal expected queue lengths.

The polynomial expression for Q_j presented in equation 5-10 is quite useful for computational purposes and also makes the association between relative saturation and expected queue length immediately apparent. However it is necessary to develop an alternative representation for Q_j in order to expedite some of the computations in Chapter 6. Note first that for $1 \leq j \leq L$,

$$A_j^k - A_j^{k+1} = \sum_{\substack{\sum_{i=1}^{L} n_i \leq N \;\&\; n_j = k}} P(n_o, n_1, \ldots, n_L)$$

Thus $$k(A_j^k - A_j^{k+1}) = n_j \sum_{\substack{\sum_{i=1}^{L} n_i \leq N \;\&\; n_j = k}} P(n_o, n_1, \ldots, n_L)$$

Also $$A_j^N = \sum_{\substack{\sum_{i=1}^{L} n_i \leq N \;\&\; n_j = N}} P(n_o, n_1, \ldots, n_L)$$

so that $$N \cdot A_j^N = n_j \sum_{\substack{\sum_{i=1}^{L} n_i \leq N \;\&\; n_j = N}} P(n_o, n_1, \ldots, n_L)$$

Substituting into equation 5-9 ,

$$Q_j = \sum_{k=0}^{N} n_j \sum_{\substack{\sum_{i=1}^{L} n_i \leq N \;\&\; n_j = k}} P(n_o, n_1, \ldots, n_L)$$

$$= \sum_{\substack{\sum_{i=1}^{L} n_j \leq N}} n_j \, P(n_o, n_1, \ldots, n_L)$$

$$= \frac{1}{G(N)} \sum_{\substack{\text{L} \\ \sum_{i=1} n_i \leq N}} n_j \prod_{i=1}^{L} (p_i u_o / u_i)^{n_i} \qquad 5\text{-}11$$

This alternative representation for Q_j will be exploited further in Chapter 6.

System Performance

Since central server networks are being considered primarily as models of batch processing systems, it is natural to define system performance in terms of the average number of jobs processed per unit time. This quantity is comparatively easy to compute for any central server network. Recall that if a central server network is observed for an interval of time of length T, the expected amount of time that the CPU will be active during this interval is $A_o T$. Since the expected amount of CPU processing time per program is $1/(u_o p_o)$ by Table 4-2, the expected number of complete programs processed during the interval is $A_o T / (1/(u_o p_o)) = A_o T u_o p_o$. Hence the average number of programs processed per unit time is $A_o T u_o p_o / T = A_o u_o p_o$. This quantity, which will be known as the processing capacity of the network, will be used in subsequent sections of this thesis as the measure of system performance.

The expression for processing capacity may appear to be heavily weighted in terms of CPU performance, but in fact

this is not the case. To demonstrate this point note that the expected amount of time that the $j^{\underline{th}}$ PPU is active during an interval of length T is $A_j T$. Since the expected amount of processing time per program on the $j^{\underline{th}}$ PPU is $p_j/(u_j p_o)$ by Table 4-2, it is also possible to express the expected number of complete programs processed during interval T as $A_j T/(p_j/(u_j p_o)) = A_j T u_j p_o/p_j$. Hence the expected number of programs processed per unit time according to this analysis is $(A_j T u_j p_o/p_j)/T = A_j u_j p_o/p_j$. But $A_j u_j p_o/p_j = A_o u_o p_o$ by the Conservation Law. Thus the processing capacity of a central server network has no special connection with CPU performance and can be represented in equivalent form in terms of the performance of any other system processor.

As a final point it should be noted that under the current definition processing capacity can only be used to compare the performance of systems which are processing identical populations of programs. All the examples that will be considered in this thesis comply with this requirement.

Bottlenecks

The term "bottleneck" is generally applied to a system component whose behavior is seriously degrading the performance of an entire system. Despite the widespread use of this term, it is not immediately obvious how to measure the degradation in system performance that is due to the behavior

of an individual component. One possible approach is to calculate the effect on system performance of a small increase in the performance of the component in question. If a small increase in component performance produces a considerable increase in system performance, it would seem reasonable to conclude that the component is seriously degrading system performance and creating a bottleneck.

As the increment in component performance used for comparison purposes becomes arbitrarily small, the extent to which a particular component is creating a bottleneck will become proportional to the rate of change of system performance with respect to the performance of that component. In the case of central server networks where individual servers correspond to system components, processing rates (i.e., $u_0, u_1, \ldots, u_L$) correspond to component performance, and system performance is measured in terms of processing capacity (i.e., $A_o u_o p_o$), it follows that the extent to which the $j^{\underline{th}}$ server is creating a bottleneck is proportional to $\frac{\partial}{\partial u_j} A_o u_o p_o$. Note that A_o is being regarded as a function of $u_0, u_1, u_2, \ldots, u_L$, $p_0, p_1, p_2, \ldots, p_L$ and N as indicated in equations 5-3 and 5-4.

If $\frac{\partial}{\partial u_i} A_o u_o p_o = \frac{\partial}{\partial u_j} A_o u_o p_o$ for all $i, j \in \{0,1,2,\ldots,L\}$ then the corresponding central server network has no bottlenecks and is in some sense balanced. If on the other hand one of the $\frac{\partial}{\partial u_j} A_o u_o p_o$ is considerably larger than all the others, then the corresponding server is creating a serious

bottleneck and a small increase in its processing speed can be expected to produce a significant increase in the system's processing capacity. This should not be construed to mean that bottlenecks are always undesirable. In fact, it is sometimes advantageous to design bottlenecks into a system. The section of Chapter 6 dealing with optimal peripheral processor utilization (p. 152 ff.) illustrates precisely such a situation.

COMPUTATIONAL FORMULAS

Basic Iterative Formula

If X_j is defined as $p_j u_o / u_j$ then equation 5-3 can be written more simply as

$$G(N) = \sum_{\sum_{j=1}^{L} n_j \leq N} \prod_{j=1}^{L} (X_j)^{n_j} \qquad 5\text{-}12$$

Equation 5-12 has an appealing mathematical symmetry and is also well suited for certain types of symbolic manipulation such as symbolic differentiation. However the computational aspects of equation 5-12 are most unattractive, especially in light of the observation* that there are $\frac{(L+N)!}{L!N!}$ states of the form $(n_o, n_1, \ldots, n_L)$ for which $\sum_{j=1}^{L} n_j \leq N$. Thus the calculation of G(N) for the comparatively modest case in which $L = 7$ and $N = 17$ requires the summation of 346,104 terms, each of which is the product of seven factors which are themselves powers of the basic units (i.e., the X_j's).

While such computations are well within the capability of modern digital computers, the large number of floating point additions is a cause for at least some concern. There is also the danger of floating point overflow since powers of X_j as high as $(X_j)^{17}$ must be calculated.

---

*This fact is demonstrated by Feller (32, p. 38).

Fortunately there exists an extremely efficient computational algorithm for evaluating G(N). For the case in which $L = 7$ and $N = 17$ this algorithm reduces the required computation to 119 additions and 119 multiplications. Furthermore, the values of $G(1),G(2),\ldots,G(16)$ are generated as intermediate results so that it is possible to proceed directly to the calculation of the marginal distribution of customers at each server once G(17) is obtained. That is, once $G(1),G(2),\ldots,G(17)$ are obtained, equation 5-7 can be used to obtain $A_o^1,A_o^2,\ldots,A_o^{17}$. The values of the A_j^k for $j=1,2,\ldots,L$ can then be obtained using the Generalized Conservation Law (i.e., equation 5-8). It is also possible to calculate expected queue lengths at this point, either directly from the G(k)'s using equation 5-10 or indirectly from the A_j^k's using equation 5-9, line 3.

Before discussing the computational algorithm for G(N) it is necessary to define one auxiliary function. Assume that $X_1,X_2,\ldots,X_L$ are specified and define

$$g(n,\ell) = \sum_{\sum_{j=1}^{\ell} n_j \leq n} \prod_{j=1}^{\ell} (X_j)^{n_j} \qquad 5\text{-}13$$

Equation 5-13 is defined for $1 \leq \ell \leq L$ and $n \geq 0$. Note that G(n) as defined in equation 5-12 is equal to $g(n,L)$ for any value of n. Note also that $g(0,\ell) = 1$ for $1 \leq \ell \leq L$.

Next note that if $n \geq 1$ and $\ell \geq 2$, then

$$g(n,\ell) = \sum_{\substack{\ell \\ \sum_{j=1} n_j \leq n \ \& \ n_\ell = 0}} \prod_{j=1}^{\ell} (X_j)^{n_j} \quad + \quad \sum_{\substack{\ell \\ \sum_{j=1} n_j \leq n \ \& \ n_\ell \geq 1}} \prod_{j=1}^{\ell} (X_j)^{n_j}$$

Now

$$\sum_{\substack{\ell \\ \sum_{j=1} n_j \leq n \ \& \ n_\ell = 0}} \prod_{j=1}^{\ell} (X_j)^{n_j} = \sum_{\substack{\ell-1 \\ \sum_{j=1} n_j \leq n}} \prod_{j=1}^{\ell-1} (X_j)^{n_j}$$

$$= g(n,\ell-1)$$

Also

$$\sum_{\substack{\ell \\ \sum_{j=1} n_j \leq n \ \& \ n_\ell \geq 1}} \prod_{j=1}^{\ell} (X_j)^{n_j} = X_\ell \sum_{\substack{\ell \\ \sum_{j=1} n_j \leq n-1}} \prod_{j=1}^{\ell} (X_j)^{n_j}$$

$$= X_\ell \, g(n-1,\ell)$$

Thus $$g(n,\ell) = g(n,\ell-1) + X_\ell \, g(n-1,\ell) \qquad 5\text{-}14$$

The boundary condition corresponding to $\ell = 1$ is

$$g(n,1) = \sum_{k=0}^{n} (X_1)^k \qquad 5\text{-}15$$

Equation 5-14 together with boundary condition 5-15 completely defines the computational algorithm for G(N). This algorithm is represented schematically in Table 5-1.

	X_1	X_2	X_3	...	X_ℓ	...	X_L
	1	2	3	...	ℓ	...	L
0	1	1	1	...	1	...	1
1	$1+X_1$						
2	$1+X_1+X_1^2$						
⋮	⋮				$g(n-1,\ell)$ $\downarrow X_\ell$		
n					$g(n,\ell-1) \longrightarrow g(n,\ell)$		
⋮	⋮						
N	$\sum_{k=0}^{N} (X_1)^k$						$g(N,L)$

Table 5-1

Algorithm Operation

Table 5-1 illustrates that each interior value of $g(n,\ell)$ is obtained by adding together the value immediately to the left of $g(n,\ell)$ (i.e., $g(n,\ell-1)$) and the value immediately above $g(n,\ell)$ multiplied by the corresponding column variable (i.e., $X_\ell \cdot g(n-1,\ell)$). Observe that the leftmost column will be properly initialized if it is assumed that there is a column of 1's immediately to the left of that column at the start of the algorithm.

Note that the ultimate objective of the algorithm is to determine the value in the lower right-hand corner of the table since this corresponds to $g(N,L) = G(N)$. However the entire rightmost column is of interest since $g(n,L) = G(n)$ for $n=1,2,\ldots,N$. Thus the values of $G(n)$ for $n=1,2,\ldots,N-1$ are natural by-products of the computation of $G(N)$.

Table 5-1 is slightly misleading since it creates the impression that it is necessary to store the entire N by L matrix of values of $g(n,\ell)$ in order to obtain the values of interest in the rightmost column. In fact it is never necessary to store more than N values at any given time.

To see this, suppose that the iteration begins with the cell in the upper left-hand corner of the table and then proceeds by moving down one column at a time. At any given instant the only values required to complete a column are those values which are below the most recently computed value and one column to the left plus of course the most recently computed value itself. In addition, all the values above the most recently computed value must be retained since they will be required in the computations for the next column. This state of affairs is represented schematically in Table 5-2.

As illustrated in Table 5-2, the basic iterative step in the algorithm involves replacing $C(n)$ by $C(n)+X_\ell C(n-1)$ and then either incrementing n by one if $n<N$ or resetting n to one and moving to the next column if $n=N$. Note that

when the algorithm terminates the final values of C(1),C(2), ...,C(N) will correspond precisely to the values in the rightmost column of Table 5-1 (i.e., to g(1,L), g(2,L), ...,g(N,L)).

	X_1	X_2	X_3	...	X_ℓ	...	X_L	
	1	2	3	...	ℓ	...	L	
0	1	1	1	...	1	...	1	
1					C(1)			
2					C(2)			
3					C(3)			
⋮					⋮			
n-1					C(n-1)	⟵	last value obtained	
n				C(n)	☐	⟵	next value to be obtained	
n+1				C(n+1)			C(n) will be set equal to	
							$C(n) + X_\ell C(n-1)$	
⋮				⋮				
N				C(N)				

Table 5-2

Storage Allocation

To implement the algorithm it is necessary to first set C(n) equal to 1 for n=0,1,...,L so that the leftmost column will be properly initialized during the first iteration. Then the basic iterative step must be carried out for each cell in the table. The complete algorithm for computing the rightmost column of Table 5-1 can thus be expressed in FORTRAN-like notation as follows:

```
      DO 1 n=0,N
    1 C(n)=1
C
      DO 2 ℓ=1,L
      DO 2 n=1,N
    2 C(n)=C(n) + X_ℓ*C(n-1)
```

Note that each evaluation of C(n) requires one addition and one multiplication. Since C(n) is evaluated a total of N·L times during the course of the algorithm, N·L additions and N·L multiplications are required for the determination of G(1),G(2),...,G(N).

The preceding example illustrates that the algorithm defined by equation 5-14 is not only efficient from a computational standpoint, but also from the standpoint of storage requirements for both data and procedure. The next section discusses the way in which this algorithm and its variants can be applied to the wide class of queueing

networks considered in Appendix B.

Extensions

Two extensions to the basic iterative formula will be considered in this section. The first extension, which is relatively minor, will cover the case of closed queueing networks with simple exponential servers. These networks are discussed in the first section of Appendix B. The second extension will cover the networks discussed in the second section of Appendix B: namely, closed queueing networks with queue dependent exponential servers.

The steady state distribution of customers in a closed queueing network with simple exponential servers is presented in equations B-8 and B-9. These equations are of the same form as equations 5-2 and 5-3 except that $y_o/u_o = 1$ in the case of 5-2 and 5-3. Setting X_j equal to y_j/u_j, equation B-8 becomes

$$P(n_o,n_1,\ldots,n_L) = \frac{1}{G(N)} \prod_{j=0}^{L} (X_j)^{n_j} \qquad 5\text{-}16$$

and equation B-9 becomes

$$G(N) = \sum_{\sum_{j=0}^{L} n_j = N} \prod_{j=0}^{L} (X_j)^{n_j} \qquad 5\text{-}17$$

Equation 5-17 can be evaluated in an efficient manner with the aid of a minor change in the definition of $g(n,\ell)$.

$$\text{Let} \quad g(n,\ell) = \sum_{\substack{\ell \\ \sum_{j=0} n_j = n}} \prod_{j=0}^{\ell} (X_j)^{n_j} \qquad 5\text{-}18$$

Next note that the argument leading from 5-13 to 5-14 is still valid and that

$$g(n,\ell) = g(n,\ell-1) + X_\ell \cdot g(n-1,\ell) \qquad 5\text{-}19$$

for $1 \leq n \leq N$ and $1 \leq \ell \leq L$.

The boundary condition corresponding to $\ell=0$ is

$$g(n,0) = (X_o)^n \qquad 5\text{-}20$$

Thus the only differences between the computation of G(N) in 5-17 and the computation of G(N) in 5-12 are the boundary condition and the presence of X_o. The boundary condition can clearly be accounted for by initializing C(n) to 0 instead of 1 for n=1,2,...,N. C(0) must still be initialized to 1. The computational algorithm for obtaining the values which correspond to the rightmost column of Table 5-1 for the case where G(N) is defined by equation 5-17 can thus be expressed in FORTRAN-like notation as follows:

```
      C(0)=1.0
      DO 1 n=1,N
    1 C(n)=0
C
      DO 2 ℓ=0,L
      DO 2 n=1,N
    2 C(n)=C(n) + X_ℓ*C(n-1)
```

The computation of marginal distributions is also quite similar to the previous case. Note that a natural analog to the Generalized Conservation Law (i.e., equation 5-8) can be derived since

$$A_j^k = \sum_{\substack{\sum_{i=0}^{L} n_i = N \;\&\; n_j \geq k}} P(n_o, n_1, \ldots, n_L) \qquad 5\text{-}21$$

$$= \sum_{\substack{\sum_{i=0}^{L} n_i = N \;\&\; n_j \geq k}} \frac{1}{G(N)} \prod_{i=0}^{L} (X_i)^{n_i} \qquad \text{by } 5\text{-}16$$

$$= \frac{1}{G(N)} (X_j)^k \sum_{\substack{\sum_{i=0}^{L} n_i = N-k}} \prod_{i=0}^{L} (X_i)^{n_i}$$

Thus $$A_j^k = (X_j)^k \frac{G(N-k)}{G(N)} \qquad 5\text{-}22$$

Equation 5-22 is valid for j=0 as well as for j=1,2,...,L. Thus, once G(1),G(2),...,G(N) are calculated, the marginal distribution of customers at each server may be readily obtained.

The extension to the case of queue dependent exponential servers is slightly more complex. If X_j is once again set equal to y_j/u_j, then equations B-15 and B-16 which define the steady state distribution become

$$P(n_o,n_1,\ldots,n_L) = \frac{1}{G(N)} \prod_{j=0}^{L} \frac{(X_j)^{n_j}}{A_j(n_j)} \qquad 5\text{-}23$$

and

$$G(N) = \sum_{\sum_{j=0}^{L} n_j = N} \prod_{j=0}^{L} \frac{(X_j)^{n_j}}{A_j(n_j)} \qquad 5\text{-}24$$

where A_j is defined in equation B-11.

In this case $g(n,\ell)$ will be redefined as

$$g(n,\ell) = \sum_{\sum_{j=0}^{\ell} n_j = n} \prod_{j=0}^{\ell} \frac{(X_j)^{n_j}}{A_j(n_j)} \qquad 5\text{-}25$$

It is assumed in equation 5-25 that $0 \leq \ell \leq L$ and $n \geq 0$. Note that it is again true that $g(0,\ell) = 1$ since $A_j(0) = 1$ by equation B-11.

Next note that if $1 \leq \ell \leq L$

$$g(n,\ell) = \sum_{k=0}^{n} \left[\sum_{\substack{\sum_{j=0}^{\ell} n_j = n \;\&\; n_\ell = k}} \prod_{j=0}^{\ell} \frac{(X_j)^{n_j}}{A_j(n_j)} \right]$$

$$= \sum_{k=0}^{n} \frac{(X_\ell)^k}{A_\ell(k)} \left[\sum_{\substack{\sum_{j=0}^{\ell} n_j = n-k \;\&\; n_\ell = 0}} \prod_{j=0}^{\ell} \frac{(X_j)^{n_j}}{A_j(n_j)} \right]$$

$$= \sum_{k=0}^{n} \frac{(X_\ell)^k}{A_\ell(k)} g(n-k,\ell-1) \qquad 5\text{-}26$$

It also immediately follows from 5-25 that the boundary condition corresponding to $\ell = 0$ is

$$g(n,0) = \frac{(X_o)^n}{A_o(n)} \qquad 5\text{-}27$$

The computational formula for $g(n,\ell)$ given in equation 5-26 is represented schematically in Table 5-3.

	X_1	X_2	X_3	...			X_ℓ	...	X_L
	1	2	3	...			ℓ	...	L
0	1	1	1	...	$1 \times \frac{(X_\ell)^n}{A_\ell(n)}$	+	1	...	1
1					$g(1,\ell-1) \times \frac{(X_\ell)^{n-1}}{A_\ell(n-1)}$	+			
2					$g(2,\ell-1) \times \frac{(X_\ell)^{n-2}}{A_\ell(n-2)}$	+			
⋮					⋮				
n-2					$g(n-2,\ell-1) \times \frac{(X_\ell)^2}{A_\ell(2)}$	+			
n-1					$g(n-1,\ell-1) \times \frac{(X_\ell)^1}{A_\ell(1)}$	+			
n					$g(n,\ell-1) \times \frac{(X_\ell)^0}{A_\ell(0)}$	+ → $g(n,\ell)$			
⋮									
N									

Table 5-3

Algorithm Operation for Queue Dependent Servers

The storage allocation policy depicted in Table 5-2 is clearly not adequate in this case since it is necessary to save the entire ℓ-1st column until the last entry in the ℓ^{th} column has been calculated. If the entries in the ℓ^{th} (i.e., current) column are represented by C(n,LC) for n=1,2,...,N and the entries in the ℓ-1st (i.e., previous) column are represented by C(n,LP) for n=1,2,...,N , then the basic iterative step of the algorithm involves setting C(n,LC) equal to $\sum_{k=0}^{n} \left[C(n-k,LP)*(X_\ell)^k \right] \Big/ A_\ell(k)$.

When expressing this algorithm as a FORTRAN-like program it is convenient to assume that C is a doubly subscripted variable with dimension N+1 by 2. The algorithm is then:

```
      C(0,1)=1                                          :
                                                        : Initialize
      DO 1 n=1,N                                        : first
                                                        : column
    1 C(n,1)=0                                          :
C
      LP=1                                              :
                                                        : Initialize
      LC=2                                              : LP and LC
C
      DO 3 ℓ=0,L
      DO 2 n=1,N
      C(n,LC)=0                                         :
                                                        : Perform
      DO 2 k=0,n                                        : basic
                                                        : iterative
    2 C(n,LC)=C(n,LC) + C(n-k,LP)*(X_ℓ**k)/A_ℓ(k)       : step
      LP=3-LP                                           :
                                                        : Interchange
    3 LC=3-LC                                           : LP and LC
```

Note that each time a column is completed LC and LP are interchanged so that the most recently computed column becomes the previous column for the next iteration and the other column becomes the storage area into which the results of the next iteration will be placed. When the algorithm terminates the values of C(n,LP) will correspond to the values of g(n,L) for n=1,2,...,N.

The marginal distribution of customers at each server is also more complicated in this case. To demonstrate how this distribution can be computed, first define

$$E_j^k = \sum_{\substack{\sum_{i=0}^{L} n_i = N \;\&\; n_j = k}} P(n_0, n_1, \ldots, n_L) \qquad 5\text{-}28$$

Note that $E_j^k = A_j^k - A_j^{k+1}$ where A_j^k is defined in equation 5-21. Thus far the strategy has been to obtain the values of A_j^k first and thus, by implication, the values of E_j^k, but in this case it is easier to obtain the values of E_j^k directly.

At this point it is necessary to introduce one additional auxiliary function. Let

$$h(n,\ell) = \sum_{\substack{\sum_{j=0}^{L} n_j = n \;\&\; n_\ell = 0}} \prod_{j=0}^{L} \frac{(X_j)^{n_j}}{A_j(n_j)} \qquad 5\text{-}29$$

Note that equation 5-29 bears a marked resemblance to equation 5-25; in fact, it is easy to see that $h(n,L) = g(n,L-1)$ for $n=0,1,\ldots,N$.

Returning to the calculation of E_j^k, note that

$$E_j^k = \sum_{\substack{\sum_{i=0}^{L} n_i = N \;\&\; n_j = k}} \frac{1}{G(N)} \prod_{i=0}^{L} \frac{(X_i)^{n_i}}{A_i(n_i)}$$

$$= \frac{1}{G(N)} \frac{(X_j)^k}{A_j(k)} \sum_{\substack{\sum_{i=0}^{L} n_i = N-k \;\&\; n_j = 0}} \prod_{i=0}^{L} \frac{(X_i)^{n_i}}{A_i(n_i)}$$

$$= \frac{(X_j)^k}{A_j(k)} \frac{h(N-k,j)}{G(N)} \qquad 5\text{-}30$$

Thus, assuming the values of $h(n,j)$ have all been calculated, the values of E_j^k can be easily obtained using equation 5-30. It has already been pointed out that the values of $h(n,L)$ are automatically calculated by the algorithm for $G(N)$ so that $h(n,L) = C(n,LC)$ for $n=0,1,\ldots,N$ at the completion of this algorithm. To obtain values of $h(n,\ell)$ for $\ell \neq L$ it is necessary to permute the sequence of X_j's so that the last (i.e., $L^{\underline{th}}$) X_j is equal to X_ℓ. The algorithm for $G(N)$ must then be applied to this permuted sequence. It is of course

possible to stop at the L-1st column in this case since the Lth column contains no new information.

It should be noted that it is possible to adopt a hybrid approach at times when some of the servers in the network are of the simple type (i.e., some of the $A_j(k)$ are identically equal to one). In these cases the values of A_j^k for the simple servers can be computed directly from equation 5-22 with no need to resort to equation 5-30. Also, if the servers are permuted so that the first S+1 are all simple, then the first S+1 columns of Table 5-3 can be computed using the algorithm of Table 5-1. The remaining L-S columns must then be computed using the more complex algorithm of Table 5-3. The hybrid algorithm appears on the following page. This algorithm should have wide applicability since many networks of interest contain at least a few simple exponential servers.

```
      C(0,1)=1                                          :
                                                        : Initialize
      DO 1 n=1,N                                        : first
                                                        : column
    1 C(n,1)=0                                          :
C
      DO 2 ℓ=0,S                                        : Evaluate first
                                                        : S+1 columns
      DO 2 n=1,N                                        : using algorithm
                                                        : for simple expo-
    2 C(n,1)=C(n,1) + Xℓ*C(n-1,1)                       : nential servers
C
      LP=1                                              :
                                                        : Initialize
      LC=2                                              : LP and LC
C
      DO 4 ℓ=S+1,L
      DO 3 n=1,N
      C(n,LC)=0                                         : Perform
                                                        : basic
      DO 3 k=0,n                                        : iterative
    3 C(n,LC)=C(n,LC) + C(n-k,LP)*(Xℓ**k)/Aℓ(k)         : step
C
      LP=3-LP                                           :
                                                        : Interchange
    4 LC=3-LC                                           : LP and LC
```

Hybrid Algorithm

for the computation of g(N,L)

CHAPTER 6: APPLICATIONS

INTRODUCTION

The analysis presented in Chapter 5 dealt primarily with the mathematical aspects of central server networks. That is, the steady state distribution and its associated properties were all derived without explicit mention of the fact that central server networks are of interest as models of multiprogramming systems. Since central server networks were treated as mathematical objects rather than mathematical models, Chapter 5 may be regarded as an excursion into the realm of pure mathematics.

In this chapter central server networks will once again be regarded as mathematical models and will be used to examine a number of problems related to the operation of actual multiprogramming systems. Three specific problems related to buffer size determination, peripheral processor utilization and page traffic balancing will be considered.

In each case the emphasis will be on gaining insight into the nature of the underlying stochastic process. While such insight has always been regarded as the primary objective of the central server model, it should be noted that the model can also be used to examine the behavior of actual multiprogramming systems simply by assigning empirically determined values to the model parameters and then correlating

predicted behavior with observed behavior. This alternative use of the central server model represents a distinct departure from the underlying theoretical orientation of this thesis and thus will not be pursued further at this time.

BUFFER SIZE DETERMINATION

Problem Definition

In order to optimize buffer size for I/O devices in a multiprogramming environment it is necessary to balance a number of interrelated factors. For example, as buffer size increases the amount of main memory space available for program storage decreases, and this in turn reduces the degree of multiprogramming and tends to degrade system performance. On the other hand, as buffer size decreases the number of I/O transfer requests per program increases. Assuming that each transfer involves a certain amount of overhead which is independent of buffer size, the total amount of overhead per program will thus increase as buffer size decreases, and this will also tend to degrade system performance. Hence it is important that buffers be neither too large nor too small.

Changes in buffer size bring about other effects as well. For example, decreasing the buffer size associated with a particular peripheral processor decreases the expected processing time per request for that processor, increases the total expected number of PPU and CPU processing requests

per program, decreases the expected time required to complete a CPU processing request (since PPU requests become more frequent), and alters the probabilities governing the selection of peripheral processors. The remainder of this section will be devoted to analyzing this set of interrelated factors with the aid of the central server model of multiprogramming.

Relation Between Buffer Size and Network Parameters

Suppose that it is desired to optimize buffer size for the 1st PPU in a central server network. Assume that the corresponding system contains M units of main memory which may be used for either program or buffer storage, and suppose that average program size excluding buffer space for the 1st PPU is equal to one memory unit.

Next suppose that the amount of time required to perform an I/O transfer on the 1st PPU is a random variable made up of two components: the first component represents overhead and has an expected value of v regardless of buffer length, while the second component represents actual transfer time and has an expected value of $s_1 \cdot b$ where s_1 is a constant and b is the length of the buffer measured in memory units. Since all processors in a central server network are assumed to have exponentially distributed service times, it will further be assumed that the amount of time necessary to carry out an I/O transfer on the 1st PPU, given that buffer size

for the 1^{st} PPU is equal to b, is an exponentially distributed random variable with mean $v + s_1 \cdot b$.

Four additional factors are required to complete the specification of the network. These are defined as follows:

r_1 = total expected amount of data transferred to the 1^{st} PPU per program (measured in memory units of data)

r_j = total expected number of processing requests directed to the j^{th} PPU per program ($j=2,3,\ldots,L$)

s_j = expected amount of processing time for a request directed to the j^{th} PPU ($j=2,3,..,L$)

C = total expected amount of CPU processing time per program

The next step is to determine the parameters of the associated central server network under the assumption that buffer size for the 1^{st} PPU is equal to b. Note first that $u_j = 1/s_j$ ($j=2,3,\ldots,L$) regardless of the value of b, and that $u_1 = 1/(v+s_1 b)$ by the definitions of v and s_1.

Determination of the branching probabilities is slightly more complicated. Since buffer size is equal to b, the total expected number of transfer requests per program directed to the 1^{st} PPU is r_1/b . Thus the total expected number of PPU processing requests per program is $r_1/b + \sum_{j=2}^{L} r_j$. Applying equation 4-4,

$$1/p_o - 1 = r_1/b + \sum_{j=2}^{L} r_j$$

Thus $$p_o = 1/(1+r_1/b+\sum_{j=2}^{L} r_j)$$

By the results collected in Table 4-2,

$$p_j/p_o = r_j \quad \text{for } j=2,3,\ldots,L$$

Thus $$p_j = r_j p_o = r_j/(1+r_1/b+\sum_{j=2}^{L} r_j)$$

Also, $$p_1/p_o = r_1/b$$

Thus $$p_1 = (r_1/b)p_o = (r_1/b)/(1+r_1/b+\sum_{j=2}^{L} r_j)$$

All the branching probabilities have now been expressed as functions of b.

To determine u_o note that $1/(u_o p_o) = C$ by Table 4-2 . Thus $u_o = 1/(p_o C) = (1+r_1/b+\sum_{j=2}^{L} r_j)/C$. The only network parameter still to be determined is N, the degree of multiprogramming. Since each program requires b memory units for a buffer for the 1st PPU and one memory unit for other purposes (including program storage), it immediately follows that $N = M/(1+b)$.

Non-Integral Values of N

This last equation introduces certain difficulties because N is no longer necessarily integral as required by the original central server model. Fortunately, there is a rather simple way to remedy this situation. Note first that one way of interpreting the statement that $N = 6.5$ is to assume that there are 6 programs in the system for half the time and 7 programs in the system the remainder of the time.

In general, the statement that $N = I + h$ where I is an integer and $0<h<1$ can be interpreted to mean that there are I programs in the system part of the time and I+1 programs in the system the rest of the time, where the fraction of the time that there are I programs in the system is equal to 1-h and the fraction of the time that there are I+1 programs in the system is equal to h.

Continuing with this line of reasoning, if $P_N(n_o,n_1,\ldots,n_L)$ is the steady state probability that there are n_j programs at the $j^{\underline{th}}$ server in a central server network, given that there are N programs in the entire system, then $P_N(n_o,n_1,\ldots,n_L)$ will be defined as $(1-h)\cdot P_I(n_o,n_1,\ldots,n_L) + h\cdot P_{I+1}(n_o,n_1,\ldots,n_L)$. That is, if N is not integral then the steady state probabilities associated with N will be defined by simple linear interpolation using the two integral values closest to N. There are obviously other ways of defining $P_N(n_o,n_1,\ldots,n_L)$ for non-integral values of N, but these will not be explored at this time since the linear interpolation method is satisfactory for the problem at hand. Chapter 7 contains a discussion of some of the alternative methods of dealing with the problem of non-integral values of N.

Optimization Equations

Now that all the network parameters have been represented as functions of b, it is possible to consider the problem of

optimizing system performance with respect to b. Since system performance is measured in terms of processing capacity, the problem is thus to optimize

$$A_o u_o p_o = \frac{\displaystyle\sum_{\substack{\sum_{j=1}^{L} n_j \leq N-1}} \prod_{j=1}^{L} (p_j u_o / u_j)^{n_j}}{\displaystyle\sum_{\substack{\sum_{j=1}^{L} n_j \leq N}} \prod_{j=1}^{L} (p_j u_o / u_j)^{n_j}} u_o p_o$$

with respect to b

where $p_o = 1/(1+r_1/b+\sum_{j=2}^{L} r_j)$

$p_1 = (r_1/b)/(1+r_1/b+\sum_{j=2}^{L} r_j)$

$p_j = r_j/(1+r_1/b+\sum_{j=2}^{L} r_j)$ for $j=2,3,\ldots,L$

$u_o = (1+r_1/b+\sum_{j=2}^{L} r_j)/C$

$u_1 = 1/(v+s_1 b)$

$u_j = 1/s_j$ for $j=2,3,\ldots,L$

and $N = M/(1+b)$ where non-integral values of N are evaluated by linear interpolation

Even though all the network parameters - except the u_j for $j=2,3,\ldots,L$ - depend on b, many of these dependencies

cancel each other out in the expression for $A_o u_o p_o$. For example, if X_j is defined as $p_j u_o / u_j$, then for $j=2,3,\ldots,L$

$$X_j = p_j u_o / u_j$$

$$= \frac{\left[r_j / (1+r_1/b+\sum_{j=2}^{L} r_j) \right] \left[(1+r_1/b+\sum_{j=2}^{L} r_j)/C \right]}{1/s_j}$$

$$= r_j s_j / C$$

Thus X_j is independent of b for $j=2,3,\ldots,L$.

Also $u_o p_o = \left[(1+r_1/b+\sum_{j=2}^{L} r_j)/C \right] \left[1/(1+r_1/b+\sum_{j=2}^{L} r_j) \right]$

$$= 1/C$$

Since $u_o p_o$ is independent of b, this factor can be omitted from the original optimization problem so that the problem becomes one of optimizing A_o - rather than $A_o u_o p_o$ - with respect to b.

Finally, note that

$$X_1 = p_1 u_o / u_1$$

$$= \frac{\left[(r_1/b)/(1+r_1/b+\sum_{j=2}^{L} r_j) \right] \left[(1+r_1/b+\sum_{j=2}^{L} r_j)/C \right]}{1/(v+s_1 b)}$$

$$= \frac{r_1}{b \cdot C}(v+s_1 b)$$

$$= \frac{r_1 v}{b \cdot C} + \frac{r_1 s_1}{C}$$

Thus the problem is to optimize

$$\frac{\displaystyle\sum_{\sum_{j=1}^{L} n_j \leq N-1} \prod_{j=1}^{L} (X_j)^{n_j}}{\displaystyle\sum_{\sum_{j=1}^{L} n_j \leq N} \prod_{j=1}^{L} (X_j)^{n_j}}$$

with respect to b

where $X_1 = \frac{r_1 s_1}{C} + \frac{r_1 v}{b \cdot C}$

$X_j = \frac{r_j s_j}{C}$ for $j=2,3,\ldots,L$

and $N = M/(1+b)$ where non-integral values of N are evaluated by linear interpolation

Analysis

At this point it is useful to consider the consequences of setting v, the expected overhead per transfer, equal to zero. In this case the only factor which remains dependent upon b is the degree of multiprogramming since $N = M/(1+b)$. This is true even though the expected number of processing requests per program for the 1st PPU (i.e., r_1/b) and the expected time to complete a processing request on the 1st PPU (i.e., $s_1 b$) still depend on b. The point is that these effects cancel each other out entirely in the zero overhead case. Hence it is desirable to select a value of

b which is as small as possible since this will maximize N and thereby optimize system performance.

The introduction of non-zero overhead significantly alters this state of affairs. When v is greater than zero a decrease in b will not only increase the expected number of processing requests per program directed to the 1st PPU (i.e., r_1/b) but also will increase the total expected overhead per program (i.e., $v \cdot r_1/b$). This increase in total overhead will tend to degrade system performance, thus counteracting the improvement in system performance which results from the increase in N.

Note that decreases in b produce two important effects: total overhead increases which tends to degrade performance, and N increases which tends to improve performance. Since the degradation associated with the first effect decreases as v decreases, it is possible to decrease b further when v is small before reaching the point at which the loss associated with the first effect outweighs the gain associated with the second effect. In other words, as v decreases the optimal buffer size also decreases.

The preceding analysis also has implications for the case in which v is held constant and the size of main memory (i.e., M) is varied. Under these circumstances the improvement associated with the second effect will be less pronounced for larger values of M since it is less important to increase N when N is already large. That is, the gain

in performance in going from N=6 to N=8 is less significant than the gain in performance in going from N=2 to N=4. Hence it is possible to decrease b further when M is small before reaching the point at which the gain associated with the second factor ceases to outweigh the loss associated with the first factor. This implies that optimal buffer size decreases as total memory size decreases.

Table 6-1 illustrates these general remarks with specific numerical examples. Each row of the table is associated with a particular value of s_1 so that, within each row, the transfer rate of the $1^{\underline{st}}$ PPU is held constant while the expected overhead per transfer (i.e., v) and the size of main memory (i.e., M) are varied. Note that within each memory group the optimal buffer size decreases as overhead decreases. It it also possible to observe that optimal buffer size decreases as main memory size decreases simply by comparing columns which have the same associated value of v.

None of the results presented thus far could be described as particularly surprising. However, Table 6-1 illustrates one effect which may indeed merit such a classification. To observe this effect it is necessary to scan down the columns of the table rather than scanning across the rows. Note that the only factor that varies within a column is the transfer rate of the $1^{\underline{st}}$ PPU. That is, the further down in a column an entry appears, the lower the associated transfer rate. A scan down any of the columns thus reveals that optimal buffer

M =	4			10			20		
v =	.25	.50	.75	.25	.50	.75	.25	.50	.75
.1	.33	.49	.65	.37	.62	.81	.38	.67	.92
.2	.33	.50	.65	.40	.66	.85	.42	.71	1.00
.3	.33	.50	.66	.43	.67	.88	.44	.78	1.03
.4	.33	.51	.66	.43	.67	.92	.49	.82	1.11
.5	.33	.51	.67	.44	.69	.96	.54	.89	1.21
.6	.33	.52	.67	.47	.73	1.00	.59	.97	1.22
.7	.34	.52	.68	.50	.76	1.00	.66	1.00	1.31
.8	.34	.53	.68	.53	.79	1.00	.70	1.08	1.41
.9	.34	.53	.69	.55	.82	1.00	.78	1.18	1.50
1.0	.35	.53	.69	.58	.85	1.00	.82	1.22	1.50
1.1	.35	.54	.70	.61	.89	1.03	.89	1.26	1.56
1.2	.35	.54	.70	.65	.92	1.06	.99	1.36	1.66
1.3	.35	.54	.71	.67	.96	1.08	1.00	1.48	1.78
1.4	.36	.55	.71	.67	1.00	1.11	1.07	1.50	1.86
1.5	.36	.55	.71	.67	1.00	1.15	1.19	1.50	1.86
s_1									

Table 6-1
Optimal Buffer Size

M = Main memory size
v = Expected overhead per transfer
s_1 = Expected time to transfer a buffer of unit size

L = 4
C = 1000
r_j = 1000 for j=1,2,3,4
s_j = 1 for j=2,3,4

} Additional problem parameters

size increases as the transfer rate decreases. In other words, slower devices should have larger buffers than faster devices.

There are no doubt factors beyond the scope of the model which tend to counteract this effect. However, it is still important to understand the factors within the model which work to bring this effect about. Recall that the two most significant consequences of changes in buffer size are the change in the degree of multiprogramming and the change in the expected amount of overhead per program associated with the 1^{st} PPU. Next note that as the transfer rate of the 1^{st} PPU decreases, the extent to which that PPU is creating a system bottleneck and degrading overall performance increases. This is true regardless of the expected overhead per transfer.

This bottleneck effect is then compounded by decreasing b since decreases in b increase the overhead associated with the 1^{st} PPU. If the 1^{st} PPU is already creating a serious bottleneck because of its low transfer rate, the additional overhead associated with small buffer size will be quite harmful, and hence it will be preferable to select a larger buffer size even though this reduces the degree of multiprogramming to a significant extent.

On the other hand, if the 1^{st} PPU has a high transfer rate and is not acting as a system bottleneck, buffers can be made quite small before the combination of overhead plus transfer rate creates a serious bottleneck. The optimal

buffer size will thus be smaller in this case since the improvement in performance due to increasing the degree of multiprogramming will not be outweighed by the bottleneck effect until b reaches a smaller value. Hence, within the context of the model, slower devices should be allocated larger buffers than faster devices.

Table 6-2 amplifies these remarks still further. The table corresponds to the row in Table 6-1 for which the time to transfer a buffer of unit size (i.e., s_1) is equal to .5 . The entries in Table 6-2 illustrate the way in which system performance varies as a function of buffer size for each combination of overhead and memory size in the corresponding row in Table 6-1. The same values appear in both Part A and Part B of Table 6-2, but the columns are grouped together differently in each part. The values in Table 6-2 are also presented graphically in Figure 6-1.

In Part A of Table 6-2 the values are grouped together according to overhead. The upper rows illustrate that, when buffer size is small, the chief factor affecting performance is the bottleneck effect created by the excessive overhead load on the 1^{st} PPU. Thus, performance is approximately the same within each overhead group even though main memory size varies greatly. In other words, the degree of multiprogramming has little effect on system performance because of the bottleneck created by the 1^{st} PPU. Note that the degradation due to the bottleneck increases as the expected overhead per

v =	.25			.50			.75		
M =	4	10	20	4	10	20	4	10	20
.1	.29	.33	.33	.18	.18	.18	.12	.13	.13
.2	.38	.53	.57	.28	.33	.33	.22	.24	.24
.3	.40	.60	.71	.33	.45	.46	.28	.33	.33
.4	.40	.62	.76	.35	.51	.57	.31	.41	.42
.5	.40	.62	.77	.36	.55	.64	.32	.46	.50
.6	.39	.62	.77	.36	.56	.68	.33	.49	.56
.7	.38	.61	.76	.35	.57	.71	.33	.51	.61
.8	.37	.60	.76	.35	.57	.71	.32	.52	.64
.9	.36	.59	.75	.34	.56	.72	.32	.53	.66
1.0	.35	.58	.74	.33	.56	.71	.32	.53	.67
1.1	.34	.57	.73	.32	.55	.71	.31	.52	.67
1.2	.33	.56	.73	.31	.54	.71	.30	.52	.68
1.3	.32	.55	.72	.31	.53	.70	.29	.51	.67
1.4	.31	.54	.71	.30	.52	.69	.29	.51	.67
1.5	.30	.53	.71	.29	.52	.69	.28	.50	.67
b									

Table 6-2 Part A

System Performance as a Function of Buffer Size

M = Main memory size

v = Expected overhead per transfer

b = Buffer size

M =	4			10			20		
v =	.25	.50	.75	.25	.50	.75	.25	.50	.75
.1	.29	.18	.12	.33	.18	.13	.33	.18	.13
.2	.38	.28	.22	.53	.33	.24	.57	.33	.24
.3	.40	.33	.28	.60	.45	.33	.71	.46	.33
.4	.40	.35	.31	.62	.51	.41	.76	.57	.42
.5	.40	.36	.32	.62	.55	.46	.77	.64	.50
.6	.39	.36	.33	.62	.56	.49	.77	.68	.56
.7	.38	.35	.33	.61	.57	.51	.76	.71	.61
.8	.37	.35	.32	.60	.57	.52	.76	.71	.64
.9	.36	.34	.32	.59	.56	.53	.75	.72	.66
1.0	.35	.33	.32	.58	.56	.53	.74	.71	.67
1.1	.34	.32	.31	.57	.55	.52	.73	.71	.67
1.2	.33	.31	.30	.56	.54	.52	.73	.71	.68
1.3	.32	.31	.29	.55	.53	.51	.72	.71	.67
1.4	.31	.30	.29	.54	.52	.51	.71	.70	.67
1.5	.30	.29	.28	.53	.52	.50	.70	.69	.67
b									

Table 6-2 Part B

System Performance as a Function of Buffer Size

M = Main memory size

v = Expected overhead per transfer

b = Buffer size

transfer increases.

Conversely, the lower rows of Table 6-2 Part B illustrate that the expected overhead per transfer has no appreciable effect on system performance when buffer size is large. In this case the most important factor limiting performance is the degree of multiprogramming, and this in turn is dependent only upon the size of main memory (i.e., M).

The curves plotted in Figure 6-1 all illustrate the fact that the degradation due to excessively small buffers is considerably more severe than the degradation due to excessively large buffers. This is because the expected total overhead per program increases quite rapidly as buffer size decreases and, in fact, goes to infinity as buffer size approaches zero. Thus it is generally better to err on the side of larger than optimal buffers in cases where some uncertainty exists.

As stated earlier, there may be other factors not represented in the central server model which tend to make large buffers more desirable for fast devices. For example, in real-time systems it is important to insure against buffer overflow even though this may result in large buffers and a sub-optimal degree of multiprogramming. The point of this analysis is not to discount the importance of these other factors, but merely to introduce one additional and perhaps unexpected factor into the decision making process.

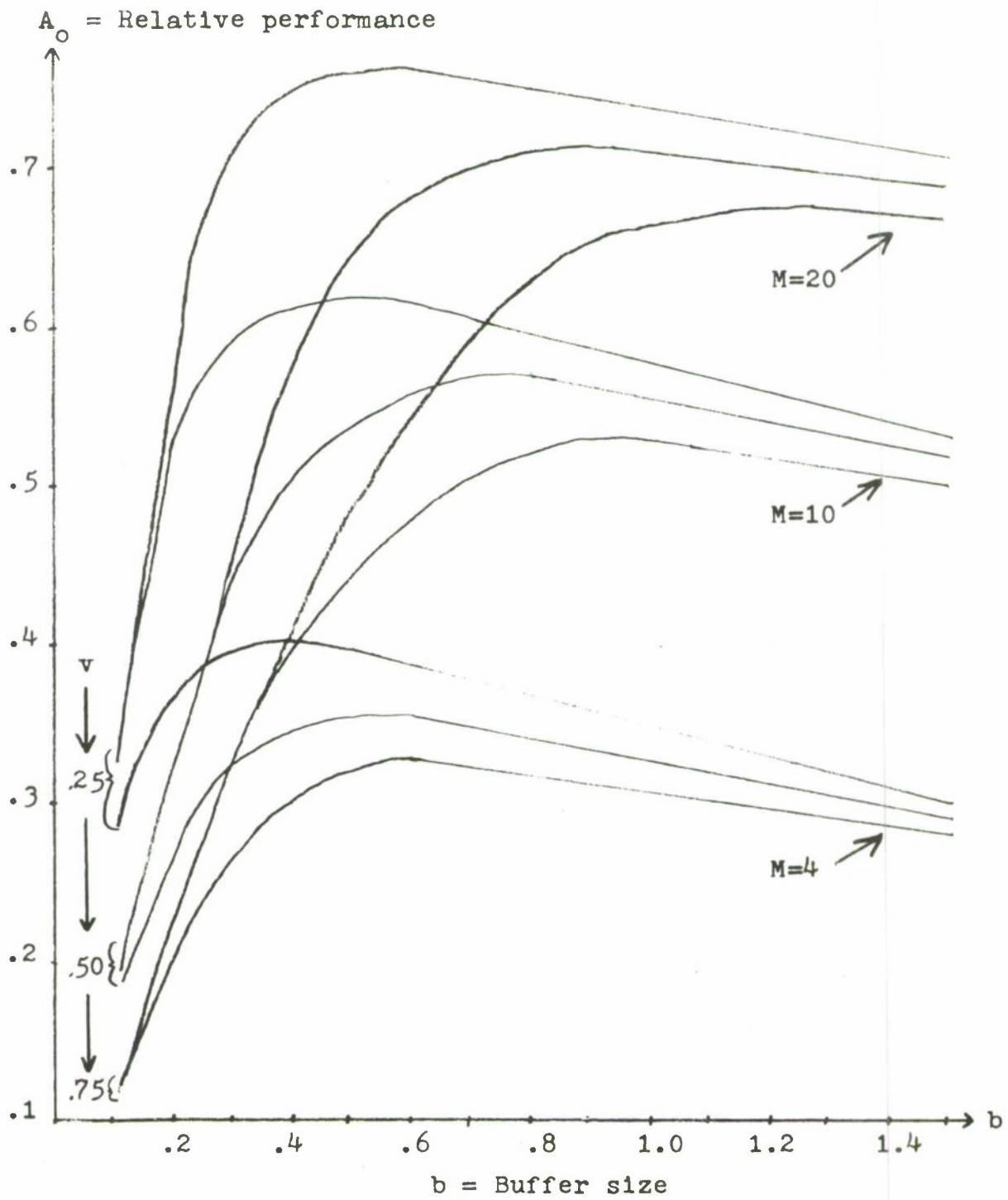


Figure 6-1

Effect of Buffer Size Variation on Relative Performance

PERIPHERAL PROCESSOR UTILIZATION

Problem Definition

Consider a system which contains a set of functionally equivalent peripheral processors such as a disk, a drum and a data cell, and assume that it is possible to vary the relative number of I/O transfer requests which are directed to each of these processors. Such variation might be brought about by altering monitor tables which control the movement of overlay segments and temporary files, or by adjusting pricing policies so that it is more economical to use one device rather than another. In any event it will be assumed that the total number of I/O transfer requests directed to this set of functionally equivalent processors is constant, but that the relative number directed to each individual processor is a specifiable parameter. The optimal selection of these parameters is thus one of the many problems of operating system design.

Optimization Equations

One method of approaching this problem is to assume that the system in question is represented by a central server network. Let S be the set which contains the subscripts of the functionally equivalent processors, and let T be the total expected number of I/O transfer requests per program directed to this set of processors. Since the

expected number of processing requests directed to the $i^{\underline{th}}$ PPU in a central server network is p_i/p_o , the values of p_i for $i \in S$ must be chosen so that $\sum_{i \in S} p_i/p_o = T$. The other parameters characterizing the network such as the speed of processors, the degree of multiprogramming and the branching probabilities for the other processors in the network (i.e., the p_i for $i \notin S$) are all constants. Assuming system performance is measured by processing capacity, the problem then is to maximize

$$A_o u_o p_o = \frac{\sum_{\sum_{j=1}^{L} n_j \leq N-1} \prod_{j=1}^{L} (p_j u_o / u_j)^{n_j}}{\sum_{\sum_{j=1}^{L} n_j \leq N} \prod_{j=1}^{L} (p_j u_o / u_j)^{n_j}} u_o p_o \qquad 6\text{-}1$$

with respect to $\{p_i \mid i \in S\}$,

subject to the constraints that $\sum_{i \in S} p_i = p_o T$ and $p_i \geq 0$.

Discussion of Results

Problems of this type fall within the realm of the calculus of variations and are generally treated using the method of Lagrange multipliers. Unfortunately this method has not yielded a closed form solution which expresses the values of p_i for $i \in S$ in terms of the other network parameters and T. However, a number of interesting relations

have been shown to hold at the point of optimal system performance.

Before discussing these relations it is worthwhile to examine the general problem on an intuitive basis. Note first that if two processors in the functionally equivalent group have the same speed (i.e., $u_i=u_j$ for $i,j \in S$), it is reasonable to expect that p_i should equal p_j at the point of optimal system performance since directing a greater proportion of processing requests to one of these processors would tend to overload it while underloading the other. This intuitive judgment has been substantiated analytically.

On the other hand there is a seemingly obvious generalization of this line of reasoning which is not valid. Suppose that processor i is t times faster than processor j (i.e., $u_i=tu_j$ for $i,j \in S$ with $t>1$). It might then seem optimal to channel t times as many processing requests to processor i as to processor j (i.e., to set $p_i=tp_j$). This would have the effect of equalizing p_i/u_i and p_j/u_j, which would then imply that $A_i = A_j$ by the Conservation Law and $Q_i = Q_j$ by equation 5-10. While these conditions may seem compatible with optimal performance, it can be shown that system performance is never optimized when $p_i=tp_j$; instead, it is preferable to have $p_i > tp_j$. That is, faster processors should receive more than their proportional share of processing requests. It then follows that faster processors should be more highly utilized and should have longer expected queues; it is also

true that performance is optimized when faster processors are creating system bottlenecks in the sense of the previous chapter.

Table 6-3 illustrates these remarks with specific numerical examples. The table deals with a central server network containing four peripheral processors, two of which are assumed to be functionally equivalent. That is, L=4 and $S = \{3,4\}$. In addition T=2, $p_o=p_1=p_2=.2$, $u_1=u_2=1$ and $u_o=5$. With these parameters held fixed, the speeds of the functionally equivalent processors (i.e., u_3 and u_4) were varied subject to the constraint that $u_3+u_4=2$. For each value of u_3/u_4 the optimal values of p_3 and p_4 were obtained by a numerical search. Each time an optimal point was located, a record was made of the following three system characteristics: p_3/p_4 , Q_3/Q_4 and $\frac{\partial}{\partial u_3} A_o u_o p_o / \frac{\partial}{\partial u_4} A_o u_o p_o$. This entire procedure was carried out for four different values of N as indicated in the table.

The table illustrates that the optimal value of p_3/p_4 is always greater than the corresponding value of u_3/u_4 as long as $u_3/u_4 > 1$. However, this effect is more noticeable for large values of u_3/u_4 and for small values of N. That is, as the difference in the speeds of the functionally equivalent processors increases or the degree of multiprogramming decreases, it becomes even more important to channel a greater number of requests to the faster processor. The ratio of expected queue lengths and the extent to which the

	p_3/p_4				Q_3/Q_4				$\frac{\partial}{\partial u_3} A_o u_o p_o \Big/ \frac{\partial}{\partial u_4} A_o u_o p_o$			
N	4	5	8	12	4	5	8	12	4	5	8	12
u_3/u_4												
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.5	2.3	2.0	1.8	1.7	1.9	1.7	1.5	1.3	1.5	1.4	1.2	1.1
2.0	4.3	3.5	2.7	2.4	3.0	2.5	1.9	1.7	2.2	1.8	1.4	1.2
2.5	7.7	5.6	3.8	3.3	4.8	3.5	2.4	2.0	3.1	2.2	1.5	1.3
3.0	14.1	8.4	5.1	4.2	7.8	4.8	2.9	2.4	4.7	2.8	1.7	1.4
3.5	28.8	12.8	6.7	5.2	14.6	6.6	3.5	2.7	8.2	3.7	1.9	1.5
4.0	98.9	19.8	8.4	6.2	46.5	9.4	4.1	3.0	24.7	4.9	2.1	1.6
4.5		32.9	10.5	7.4		14.6	4.8	3.4		7.3	2.3	1.6
5.0		64.0	13.0	8.7		26.8	5.5	3.7		12.8	2.6	1.7

Table 6-3

System Characteristics at Points of Optimal Performance

faster processor is creating a bottleneck are also seen to be positively correlated with the difference in the speeds of the two processors and negatively correlated with the degree of multiprogramming.

These observations can be explained on an intuitive basis by noting that it is preferable to channel processing requests to the faster PPU as long as the expected waiting time plus processing time for the faster PPU is less than the expected waiting time plus processing time for the slower PPU. As the degree of multiprogramming decreases, the expected waiting time for all processors in the system decreases, and hence it becomes possible to channel more and more processing requests to the faster PPU before the build up in waiting time overtakes the processing speed advantage. To further illustrate this point, note that in the limiting case where the degree of multiprogramming is equal to one, the faster PPU should receive all the processing requests since there is never any queue wait. This analysis does not take into account limits in storage capacity or other factors which might make it necessary to channel at least some requests to the slower PPU in this case.

Increasing the speed differential between the faster PPU and the slower PPU reduces the expected waiting and processing time for the former while increasing the expected waiting and processing time for the latter. Thus, to maintain optimal performance in this case, it is necessary to

increase the expected queue length at the faster PPU by channeling still more processing requests to it. Hence, on intuitive grounds it can be seen that the optimal proportion of processing request channeled to the faster PPU increases when either the degree of multiprogramming decreases or the speed differential between the PPU's increases.

Mathematical Analysis

Two useful conditions will be shown to hold at the point of optimal system performance. The first is that

$$\frac{p_i}{p_j} = \frac{\sum_{k=1}^{N} c_k (p_i u_o / u_i)^k}{\sum_{k=1}^{N} c_k (p_j u_o / u_j)^k} \qquad \text{for all } i, j \in S. \qquad 6\text{-}2$$

The coefficients c_k in equation 6-2 are defined as follows:

$$c_k = \frac{G(N-k)}{G(N)} - \frac{G(N-1-k)}{G(N-1)} \qquad \text{for } k=1,2,\ldots,N-1$$

and $c_N = 1/G(N)$

Equation 6-2 implies that $p_i/p_j = u_i/u_j$ at the point of optimal performance if and only if $u_i = u_j$. To see this, note that $p_i/p_j = u_i/u_j$ implies $p_i u_o/u_i = p_j u_o/u_j$, which in turn implies that the right hand side of equation 6-2 is equal to one. Thus, if $p_i/p_j = u_i/u_j$ and $u_i = u_j$, then equation 6-2 will be satisfied and the system will be optimized. Conversely, if $p_i/p_j = u_i/u_j$ and the system is optimized (i.e., equation 6-2 is satisfied), then the left hand side of 6-2 must be equal to one. This then implies $u_i = u_j$.

The second condition of interest which holds at the point of optimal performance is that

$$\frac{A_i}{A_j} = \frac{\frac{\partial}{\partial u_i} A_o u_o p_o}{\frac{\partial}{\partial u_j} A_o u_o p_o} \qquad 6\text{-}3$$

In addition to simplifying the computations for the last set of columns in Table 6-3, this equation also demonstrates that bottlenecks must exist at the point of optimal performance if $u_i \neq u_j$ since A_i/A_j is equal to one if and only if $p_i/p_j = u_i/u_j$ by the Conservation Law, and this equality will exist at the point of optimal performance if and only if $u_i = u_j$ as demonstrated in the previous paragraph. Thus, if $u_i \neq u_j$ there must be a bottleneck at the point of optimal performance.

Equations 6-2 and 6-3 can be formally derived from the optimization conditions for equation 6-1 by the method of Lagrange multipliers. Before applying this technique it is useful to make a few elementary observations. First note that p_o cannot be a member of the set $\{p_i \mid i \in S\}$ since the CPU is not being considered as one of the functionally equivalent processors. Hence the factor $u_o p_o$ is invariant with respect to $\{p_i \mid i \in S\}$ and may be disregarded during the optimization procedure.

Next note that $A_o = G(N-1)/G(N)$ where $G(N)$ is defined in equation 5-3 . Thus the original problem is equivalent to the problem of maximizing $G(N-1)/G(N)$ with respect to

$\{p_i \mid i \in S\}$ subject to the constraints that $\sum_{i \in S} p_i = p_o T$ and $p_i \geq 0$. Applying the method of Lagrange multipliers, this is then equivalent to maximizing

$$G(N-1)/G(N) + \alpha \left[\sum_{i \in S} p_i - p_o T \right] \qquad 6\text{-}4$$

with respect to $\{p_i \mid i \in S\}$ subject only to the constraint that $p_i \geq 0$ for all $i \in S$.*

The next step is to set all the partial derivaties of 6-4 equal to zero. For any $i \in S$,

$$\frac{\partial}{\partial p_i} \left[G(N-1)/G(N) + \alpha \left[\sum_{i \in S} p_i - p_o T \right] \right]$$

$$= \frac{\partial}{\partial p_i} G(N-1)/G(N) + \alpha$$

Now $$\frac{\partial}{\partial p_i} G(N) = \frac{\partial}{\partial p_i} \sum_{\sum_{j=1}^{L} n_j \leq N} \prod_{j=1}^{L} (p_j u_o / u_j)^{n_j}$$

$$= \sum_{\sum_{j=1}^{L} n_j \leq N} n_i (p_i u_o / u_i)^{n_i - 1} (u_o / u_i) \prod_{\substack{j=1 \\ j \neq i}}^{L} (p_j u_o / u_j)^{n_j}$$

---

*α, which is known as a Lagrange multiplier, is treated as a constant during the optimization process. After the optimal values of $\{p_i \mid i \in S\}$ are obtained (as functions of α), the value of α is selected so that the constraint $\sum_{i \in S} p_i = p_o T$ is satisfied.

$$= \frac{1}{p_i} \sum_{\substack{\sum_{j=1}^{L} n_j \leq N}} n_i \prod_{j=1}^{L} (p_j u_o / u_j)^{n_j}$$

$$= G(N) \cdot Q_i(N)/p_i$$

This last step follows from the definition of Q_i given in equation 5-11. In this discussion the dependency of Q_i on N is being explicitly represented since expressions of the form $Q_i(N-1)$ will also be needed.

Continuing with the analysis,

$$\frac{\partial}{\partial p_i} \frac{G(N-1)}{G(N)} = \frac{G(N)\ G(N-1)\ Q_i(N-1)/p_i \ - \ G(N-1)\ G(N)\ Q_i(N)/p_i}{G(N)^2}$$

$$= \frac{G(N-1)}{G(N)} \frac{1}{p_i} \left[Q_i(N-1) - Q_i(N) \right]$$

Hence the partial derivative of 6-4 with respect to p_i is

$$\frac{G(N-1)}{G(N)} \frac{1}{p_i} \left[Q_i(N-1) - Q_i(N) \right] + \alpha$$

Setting this derivative equal to zero yields

$$\alpha = \frac{G(N-1)}{G(N)} \frac{1}{p_i} \left[Q_i(N) - Q_i(N-1) \right]$$

Since all the partial derivatives are equal to zero at the point of optimal performance, the following relationship must hold for any $i, j \in S$ at this point:

$$\frac{G(N-1)}{G(N)} \frac{1}{p_i} \left[Q_i(N) - Q_i(N-1) \right] = \frac{G(N-1)}{G(N)} \frac{1}{p_j} \left[Q_j(N) - Q_j(N-1) \right]$$

Equivalently,

$$\frac{p_i}{p_j} = \frac{Q_i(N) - Q_i(N-1)}{Q_j(N) - Q_j(N-1)} \qquad 6\text{-}5$$

By equation 5-10,

$$Q_i(N) = \sum_{k=1}^{N} (p_i u_o / u_i)^k \frac{G(N-k)}{G(N)}$$

Thus

$$Q_i(N) - Q_i(N-1) = \sum_{k=1}^{N-1} (p_i u_o / u_i)^k \left[\frac{G(N-k)}{G(N)} - \frac{G(N-1-k)}{G(N-1)} \right]$$

$$+ \ (p_i u_o / u_i)^N / G(N)$$

Setting $c_N = 1/G(N)$

and $c_k = \frac{G(N-k)}{G(N)} - \frac{G(N-1-k)}{G(N-1)}$ for $k=1,2,\ldots,N-1$

yields $$\frac{p_i}{p_j} = \frac{\sum_{k=1}^{N} c_k (p_i u_o / u_i)^k}{\sum_{k=1}^{N} c_k (p_j u_o / u_j)^k}$$

This completes the derivation of equation 6-2. To derive equation 6-3, note that

$$\frac{\partial}{\partial u_i} A_o = \frac{\partial}{\partial u_i} \frac{G(N-1)}{G(N)} = \frac{G(N) \frac{\partial}{\partial u_i} G(N-1) - G(N-1) \frac{\partial}{\partial u_i} G(N)}{G(N)^2}$$

$$\text{Now } \frac{\partial}{\partial u_i} G(N) = \frac{\partial}{\partial u_i} \sum_{\substack{\sum_{j=1}^{L} n_j \le N}} \prod_{j=1}^{L} (p_j u_o / u_j)^{n_j}$$

$$= \sum_{\substack{\sum_{j=1}^{L} n_j \le N}} n_i (p_i u_o / u_i)^{n_i - 1} (-p_i u_o / u_i^2) \prod_{\substack{j=1 \\ j \neq i}}^{L} (p_j u_o / u_j)^{n_j}$$

$$= \frac{-1}{u_i} \sum_{\substack{\sum_{j=1}^{L} n_j \le N}} n_i \prod_{j=1}^{L} (p_j u_o / u_j)^{n_j}$$

$$= -G(N) \cdot Q_i(N) / u_i$$

$$\text{Thus } \frac{\partial}{\partial u_i} A_o = \frac{-G(N) \cdot G(N-1) \cdot Q_i(N-1)/u_i + G(N-1) \cdot G(N) \cdot Q_i(N)/u_i}{G(N)^2}$$

$$= \frac{G(N-1)}{G(N)} \frac{1}{u_i} \left[Q_i(N) - Q_i(N-1) \right]$$

$$= \frac{A_o}{u_i} \left[Q_i(N) - Q_i(N-1) \right]$$

This in turn implies

$$Q_i(N) - Q_i(N-1) = \frac{u_i}{A_o} \frac{\partial}{\partial u_i} A_o$$

Substituting in equation 6-5,

$$\frac{p_i}{p_j} = \frac{\frac{u_i}{A_o}\frac{\partial}{\partial u_i}A_o}{\frac{u_j}{A_o}\frac{\partial}{\partial u_j}A_o}$$

Therefore

$$\frac{A_o p_i/u_i}{A_o p_j/u_j} = \frac{\frac{\partial}{\partial u_i}A_o}{\frac{\partial}{\partial u_j}A_o}$$

But $\frac{A_o p_i/u_i}{A_o p_j/u_j} = \frac{A_o u_o p_i/u_i}{A_o u_o p_j/u_j} = \frac{A_i}{A_j}$ by the Conservation Law.

Thus $\frac{A_i}{A_j} = \frac{\frac{\partial}{\partial u_i}A_o}{\frac{\partial}{\partial u_j}A_o}$ at the point of optimal performance.

Equation 6-3 then follows immediately.

PAGE TRAFFIC BALANCING

Problem Definition

Multiprogramming systems which make use of paging provide another area of application for the central server model. An important feature of such systems is that, at any given time, some of a program's pages will reside in main memory while others reside in auxiliary memory. When a program references a page which is not in main memory, that page must be transferred in from auxiliary memory. In order to make room for this page it is sometimes necessary to first transfer a page out of main memory. These page transfers are handled by a PPU which will be called the page transfer processor.

An interesting aspect of paged systems is that it is possible to vary the number of page transfers per program by varying the total number of pages that a program is permitted to maintain in main memory at any time. That is, in systems where only a small number of pages from each program are maintained in main memory, references to pages not in main memory will be fairly frequent, and so the total number of page transfers per program will be high. On the other hand, in systems where a large number of pages from each program are maintained in main memory, references to pages not in main memory will be relatively infrequent once the initial set of pages is loaded, and so the total number of page

transfers per program will be fairly low. Thus it would appear advantageous to maintain a large number of pages from each program in main memory.

However, with the size of main memory fixed, systems which maintain a small number of pages from each program in main memory will have a large number of programs in main memory at any time, which is to say a high degree of multiprogramming. This will tend to improve system performance by enabling other programs to utilize the CPU and various PPU's while the page transfer channel is carrying out a transfer for a particular program. Hence there are also advantages to maintaining a small number of pages from each program in main memory. This suggests it should be possible to optimize system performance by specifying the number of pages which each program may maintain in main memory in a way that keeps the number of page transfers per program relatively small while allowing the degree of multiprogramming to be relatively high. This optimization problem, which will be referred to as page traffic balancing, can be partially resolved with the aid of the central server model.

Parametric Specification of Page Traffic Behavior

Before approaching this problem it is necessary to have some way of specifying the relationship between the expected number of page transfers per program and the average number of pages each program is permitted to maintain in main memory.

Note that the degree of multiprogramming is equal to the total number of pages of main memory divided by the average number of pages each program is permitted to maintain in main memory. Assuming the total number of pages of main memory is being held fixed, it is thus sufficient to express the relationship between the expected number of page transfers per program and the degree of multiprogramming of the system.

Generally speaking, the expected number of page transfers per program increases as the degree of multiprogramming increases. This increase is comparatively gradual at first, but then accelerates abruptly after the degree of multiprogramming passes a certain critical threshold. This abrupt acceleration is due to a phenomenon known as thrashing which was originally analyzed by Denning (30).

Page traffic behavior is also affected by the page replacement algorithm. This algorithm determines which page to remove from main memory at times when it is necessary to make room for a new page. A good page replacement algorithm will remove a page which is not likely to be referenced again in the near future, thus reducing unnecessary page transfers. It is not the purpose of this discussion to present the details of various page replacement algorithms or an analysis of the thrashing phenomenon, but merely to characterize these factors in a relatively simple manner which preserves their essential features and also permits systematic variation of

key parameters.

Figure 6-2 illustrates such a characterization. The curves which appear in this figure correspond to instances of equation 6-6 for which B=1, T=10 and A=0.5, 1.0 and 2.0 .

$$F(N) = B \cdot \left[\frac{T-1}{T-N}\right]^{A} \qquad 6\text{-}6$$

Equation 6-6 expresses the expected number of page transfers per program (i.e., F(N)) as a function of the degree of multiprogramming (i.e., N) and three parameters: A, B and T. The parameter T represents the degree of multiprogramming at which the thrashing phenomenon causes the expected number of page transfers per program to become virtually infinite. Since thrashing continues if N is increased beyond T, it will be assumed that equation 6-6 defines F(N) only for the case in which $1 \leq N < T$. For $N \geq T$ F(N) is assumed to be infinite.

Next note that F(N) = B when N = 1. Hence B is the expected number of page transfers per program when only one program is maintained in main memory at any time. Assuming that main memory is large enough to accomodate entire programs in this case (i.e., no overlays are necessary), B is then equal to the expected number of pages referenced per program. It is assumed in Figure 6-2 that B = 1, but by simply reinterpreting the scale along the vertical axis it is possible to represent any other value of B.

The exponent A in equation 6-6 is intended to represent the relative efficiency of various page replacement algorithms.

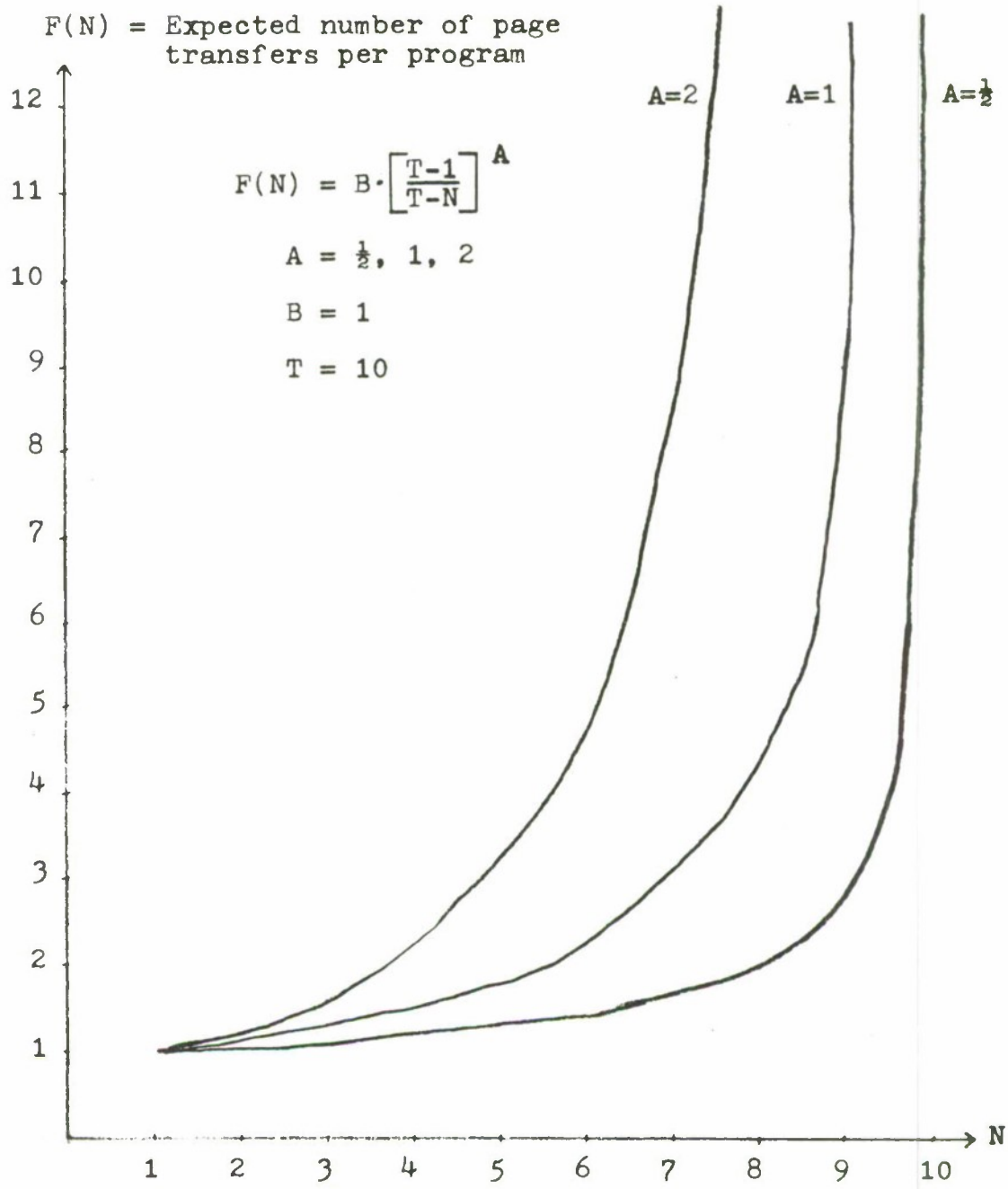


Figure 6-2

Page Traffic Behavior

If N is held constant then F(N) will decrease as A decreases. This corresponds to the fact that more efficient page replacement algorithms will result in a smaller expected number of page transfers per program when all other factors are held fixed.

No particular correspondence between specific values of A and actual page replacement algorithms is intended, although the value of A associated with relatively inefficient algorithms such as FIFO will be greater than the value associated with more efficient algorithms such as LRU. It is assumed that A is restricted to positive values (i.e., A>0). Hence the thrashing effect will always cause the expected number of page transfers per program to go to infinity as N approaches T. This corresponds to the fact that thrashing will occur in any system in which main memory is overcommitted, regardless of the page replacement algorithm used. However, Figure 6-2 illustrates that inefficient page replacement algorithms cause thrashing to become a serious problem at significantly smaller values of N.

Relation Between Page Traffic Behavior and Network Parameters

Assume that the three parameters in equation 6-6 have all been specified, and suppose that it is desired to optimize system performance with respect to N. To treat this problem using the central server model of multiprogramming,

let the page transfer processor correspond to the 1^{st} PPU in the network. In addition, assume that the following parameters have been specified:

r_j = total expected number of processing requests directed to the j^{th} PPU per program ($j=2,3,\ldots,L$)

s_j = expected amount of processing time for a request directed to the j^{th} PPU ($j=1,2,\ldots,L$)

C = total expected amount of CPU processing time per program

Note that these parameters closely correspond to those used in the buffer size determination problem, the major difference being that s_1 is independent of N in the case of page traffic balancing since the time to transfer a page to or from main memory is not assumed to be dependent upon the degree of multiprogramming of the system. Hence $u_j = 1/s_j$ for $j=1$ as well as for $j=2,3,\ldots,L$.

The other network parameters, which do depend on N, may be determined as follows. The total expected number of PPU processing requests per program is $F(N) + \sum_{j=2}^{L} r_j$. Thus

$$1/p_o - 1 = F(N) + \sum_{j=2}^{L} r_j \qquad \text{by equation 4-4 .}$$

Hence $$p_o = 1/(1+F(N)+\sum_{j=2}^{L} r_j)$$

Paralleling the argument presented in the buffer size determination problem,

$$p_j/p_o = r_j \qquad \text{for } j=2,3,\ldots,L$$

so that $$p_j = r_j p_o = r_j/(1+F(N)+\sum_{j=2}^{L} r_j)$$

Also $$p_1/p_o = F(N)$$

which implies $$p_1 = F(N)p_o = F(N)/(1+F(N)+\sum_{j=2}^{L} r_j)$$

Finally $$1/(u_o p_o) = C$$

Thus $$u_o = 1/(p_o C) = (1+F(N)+\sum_{j=2}^{L} r_j)/C$$

Optimization Equations

Summarizing the results of the previous section, the page traffic balancing problem is the problem of optimizing

$$A_o u_o p_o = \frac{\sum_{\sum_{j=1}^{L} n_j \leq N-1} \prod_{j=1}^{L} (p_j u_o/u_j)^{n_j}}{\sum_{\sum_{j=1}^{L} n_j \leq N} \prod_{j=1}^{L} (p_j u_o/u_j)^{n_j}} u_o p_o$$

with respect to N

where $$p_o = 1/(1+F(N)+\sum_{j=2}^{L} r_j)$$

$$p_1 = F(N)/(1+F(N)+\sum_{j=2}^{L} r_j)$$

$$p_j = r_j/(1+F(N)+\sum_{j=2}^{L} r_j) \qquad \text{for } j=2,3,\ldots,L$$

$$u_o = (1+F(N)+\sum_{j=2}^{L} r_j)/C$$

$$u_j = 1/s_j \qquad \text{for } j=1,2,3,\ldots,L$$

It is assumed that N is permitted to vary continuously in the interval $[1,T)$ and that non-integral values of N are evaluated by linear interpolation as in the case of buffer size determination.

Another similarity with the case of buffer size determination is that many of the dependencies on N cancel each other out in the expression for $A_o u_o p_o$. For example, with X_j again defined as $p_j u_o / u_j$, it follows that for $j=2,3,\ldots,L$

$$X_j = p_j u_o / u_j$$

$$= \frac{\left[r_j / (1+F(N)+\sum_{j=2}^{L} r_j) \right] \cdot \left[(1+F(N)+\sum_{j=2}^{L} r_j)/C \right]}{1/s_j}$$

$$= r_j s_j / C$$

$$X_1 = p_1 u_o / u_1$$

$$= \frac{\left[F(N) / (1+F(N)+\sum_{j=2}^{L} r_j) \right] \cdot \left[(1+F(N)+\sum_{j=2}^{L} r_j)/C \right]}{1/s_1}$$

$$= F(N) s_1 / C$$

Finally,

$$u_o p_o = \left[(1+F(N)+\sum_{j=2}^{L} r_j)/C \right] \cdot \left[1/(1+F(N)+\sum_{j=2}^{L} r_j) \right]$$

$$= 1/C$$

Since $u_o p_o$ is independent of N, this factor can be omitted from the original optimization problem so that the problem becomes one of optimizing A_o - rather than $A_o u_o p_o$ - with respect to N.

Thus the page traffic balancing problem is the problem of optimizing

$$\frac{\displaystyle\sum_{\sum_{j=1}^{L} n_j \leq N-1} \prod_{j=1}^{L} (X_j)^{n_j}}{\displaystyle\sum_{\sum_{j=1}^{L} n_j \leq N} \prod_{j=1}^{L} (X_j)^{n_j}}$$

with respect to N

where $X_1 = F(N)s_1/C$

$X_j = r_j s_j / C$ for $j=2,3,\ldots,L$

$F(N) = B \cdot \left[\frac{T-1}{T-N}\right]^A$

and non-integral values of N are evaluated by linear interpolation

Analysis

Assuming that the program population and the size of main memory are held constant, the two most obvious ways

to improve the performance of a paged computer system are to improve the page replacement algorithm (i.e., decrease A) or to increase the speed of the page transfer processor (i.e., decrease s_1). In order to determine the relationship between these two factors, u_1 ($u_1=1/s_1$) was allowed to vary from 0.1 to 4.0 while A was set to either 0.5, 1.0 or 2.0 . For each value of u_1 and A the optimal value of N was obtained by a numerical search procedure and the associated value of A_o was computed. The other parameters in the model were: B=1000, T=10; r_j=1000, s_j=1, C=1000; L=4.

The outcome of this optimization procedure is presented in Figure 6-3 . As anticipated, optimal performance is improved both by decreasing A and by increasing u_1. It is interesting to note that decreasing A from 2.0 to 0.5 improves performance from .137 to .167 (i.e., by 29%) when $u_1 = 0.2$, while the same change in A improves performance from .490 to .659 (i.e., by 34%) when $u_1 = 2.0$. Thus the benefits of using a better page replacement algorithm may be more significant for fast page transfer processors than for slow page transfer processors. This illustrates the point that choice of page replacement algorithm may be more - rather than less - critical as the speed of the page transfer processor increases.

This section illustrates one way in which central server models can be used to analyze the problem of page traffic balancing. There is obviously much additional work to be

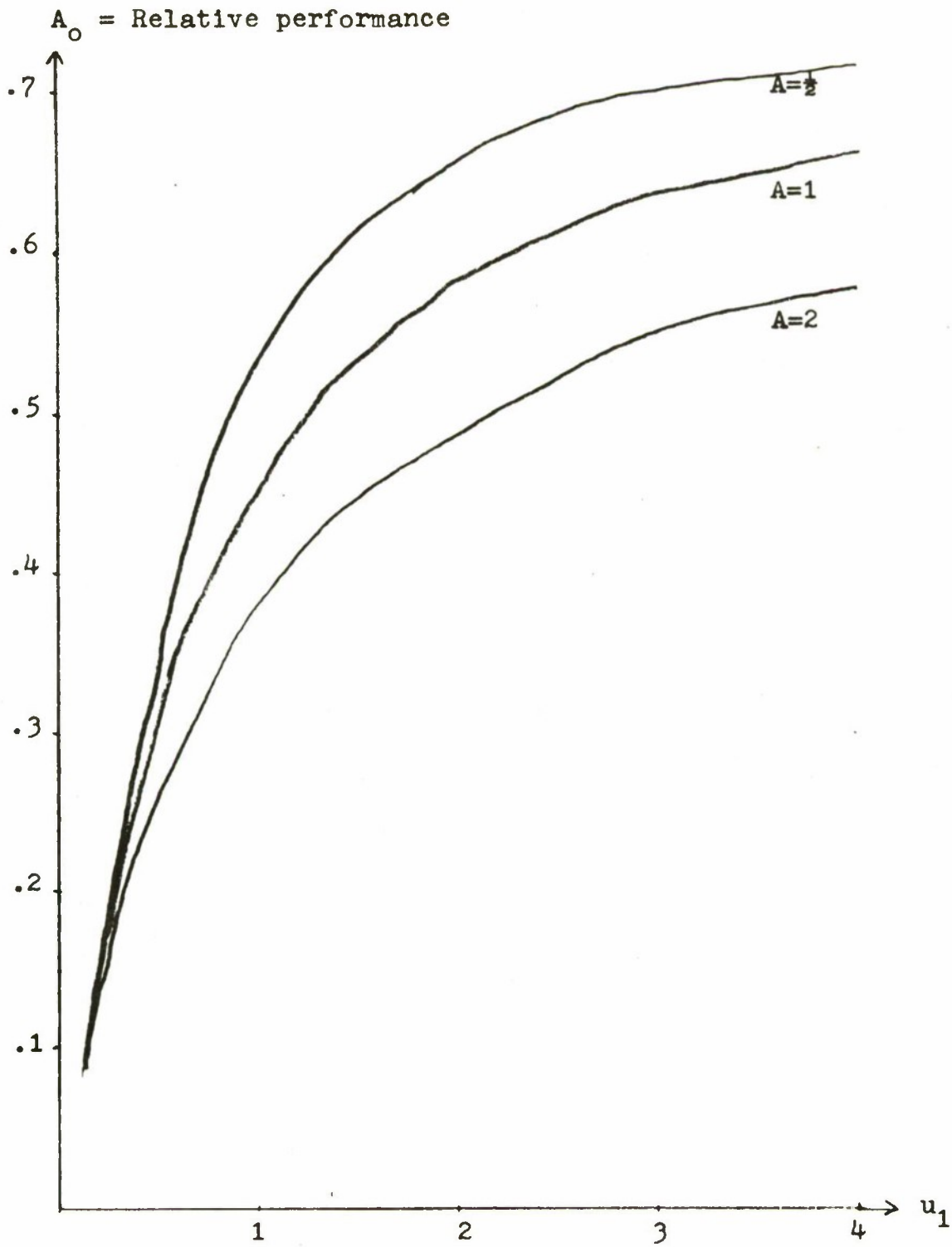


Figure 6-3

Effect of Page Replacement Algorithm and Speed of Page Transfer Processor on Relative Performance

done in this area, particularly in determining the nature of the function F(N). Note that F(N) could be determined empirically for different page replacement algorithms by direct measurement of actual systems. These empirical results could then be combined with the optimization equations of this section to explore the behavior of paged systems in considerably greater depth.

CHAPTER 7: EXTENSIONS

INTRODUCTION

The central server model of Figure 4-3 incorporates a set of general features which are common to virtually all large scale multiprogramming systems. However, when constructing models of particular systems it is sometimes desirable to extend the model by adding certain special features such as multiple CPU's, interactive time-sharing terminals and sector scheduled drums. It may also be desirable to examine the consequences of random fluctuations in the degree of multiprogramming (i.e., the value of N). This chapter discusses a number of relatively simple extensions which can be made to the basic central server model in order to incorporate features of this type.

Some of the extensions presented in this chapter are rather obvious, given the basic model of Figure 4-3 and the solution techniques developed by Jackson (48) and Gordon and Newell (41). The reason for including these extensions along with the others is to provide a compact point of reference for future work in this area. In addition the entire set of extensions serves to illustrate the generality and flexibility of the original central server model.

NEW PROCESSOR TYPES

Multiple Processors and Channels

In the diagram of Figure 4-3 each service facility (i.e., circle) is understood to represent an individual processor or server. Suppose instead that, for $j=0,1,2,\ldots,L$, the $j^{\underline{th}}$ service facility represents a set of m_j identical servers which can operate in parallel. For example, there may be a number of CPU's at the central service facility or a number of data channels associated with a set of disk drives at one of the peripheral service facilities. Thus each m_j is a positive integer which may in some cases be equal to one.

To obtain the steady state distribution for such a network assume first that the time required to complete a service request at one of the servers in the $j^{\underline{th}}$ service facility is an exponentially distributed random variable with mean $1/u_j$. It then follows immediately from equation B-15 of Appendix B that the steady state distribution is given as

$$P(n_o,n_1,\ldots,n_L) = \frac{1}{G(N)} \frac{1}{A_o(n_o)} \prod_{j=1}^{L} \frac{(p_j u_o/u_j)^{n_j}}{A_j(n_j)} \qquad 7\text{-}1$$

$$\text{where} \quad A_j(k) = \begin{cases} k! & \text{if } k \leq m_j \\ m_j!(m_j)^{k-m_j} & \text{if } k > m_j \end{cases} \qquad 7\text{-}2$$

Equation 7-1 represents an entirely straightforward application of the solution techniques developed by Jackson and Gordon and Newell. Computational algorithms for evaluating equation 7-1 appear in the second half of Chapter 5.

Dedicated Peripheral Processors

Dedicated peripheral processors correspond to devices such as the interactive terminals of a time-sharing system. As in the case of multiple processors and channels, it is assumed that all the dedicated peripheral processors of a particular type have identical service time distributions and can operate in parallel with one another. However, it is assumed that there is a dedicated processor of each type for each program in the system. Thus there are never any queueing delays associated with service requests for dedicated peripheral processors.

From a mathematical standpoint a set of dedicated peripheral processors corresponds to a service facility containing a sufficiently large number of parallel servers to guarantee that no service request ever has to wait in a queue. For a closed network of N circulating customers, N parallel servers will obviously suffice. Hence the assumption that the i^{th} service facility in a central server network corresponds to a set of dedicated peripheral servers is equivalent to the assumption that the steady state distribution of the network is given by equation 7-1 and that

$m_i = N$ in equation 7-2. Note that $m_i = N$ implies

$$A_i(k) = k! \quad \text{for all } k \qquad 7\text{-}3$$

It is convenient to use equation 7-3 to characterize dedicated peripheral processors since this equation contains no explicit reference to the value of m_i.

Even though dedicated peripheral processors and multiple processors and channels can be treated by the same mathematical techniques, they are not conceptually identical. This follows from the observation that multiple processors and channels are regarded as functionally equivalent, which is to say that each one can service a processing request from any program in the system. On the other hand, each dedicated peripheral processor is restricted to serving the processing requests of a particular program. The two concepts are mathematically equivalent because it makes no difference which processor is serving which request as long as all requests can be served in parallel.

Queue Dependent Processors

The discussion of rotating storage service disciplines presented in Chapter 2 indicates that it is possible to improve the performance of devices such as disks and drums by employing scheduling algorithms such as SATF (i.e., shortest access time first) and SSTF (i.e., shortest seek time first). Under these scheduling algorithms the expected service time

per processing request becomes a function of the number of requests waiting for service at the facility. In other words, service times become queue dependent.

Suppose that the $i^{\underline{th}}$ service facility in a central server network consists of a queue dependent processor whose service time is an exponentially distributed random variable with mean $1/(u_i \cdot a_i(n_i))$ where n_i is the number of programs at the facility and $a_i(n_i)$ is an arbitrary positive valued function. It then follows from equation B-15 that the steady state distribution for this network is given by equation 7-1 with $A_i(k)$ defined as follows:

$$A_i(k) = \begin{cases} 1 & \text{if } k = 0 \\ \prod_{n=1}^{k} a_i(n) & \text{if } k > 0 \end{cases} \qquad 7\text{-}4$$

Smith (77) presents an analysis technique which can be used to determine the function $a_i(k)$ for the case in which the $i^{\underline{th}}$ service facility is a drum employing an SATF scheduling algorithm. The references in the section of Chapter 2 dealing with rotating storage service disciplines are also relevant to this problem.

Non-Exponential Dedicated Peripheral Processors*

Suppose that the j^{th} service facility in a central server network consists of a set of dedicated peripheral processors whose service time density function is

$$h \cdot u_{j,1} e^{-u_{j,1} t} + (1-h) \cdot u_{j,2} e^{-u_{j,2} t} \qquad 7\text{-}5$$

Expression 7-5 represents a hyperexponential density function of the second degree. It is assumed that $0 < h < 1$ and $u_{j,1} \neq u_{j,2}$.

Expression 7-5 implies that the amount of service time per processing request for the j^{th} service facility is distributed as $u_{j,1} e^{-u_{j,1} t}$ with probability h and distributed as $u_{j,2} e^{-u_{j,2} t}$ with probability 1-h. Suppose that p_j is the probability that a program will generate a processing request for the j^{th} service facility after completing a CPU processing request. The j^{th} service facility may then be conceptually divided into a pair of service facilities as illustrated in Figure 7-1.

Partitioning the j^{th} service facility in this way creates a new central server network with exponential service times at all points and a state description vector of the form $(n_o, n_1, \ldots, n_{j-1}, n_{j,1}, n_{j,2}, n_{j+1}, \ldots, n_L)$. The steady

---

*The material presented in this section was suggested by C.G. Moore and S. Kimbleton of the University of Michigan. Some of this material appears in Moore's Ph.D. thesis (62).

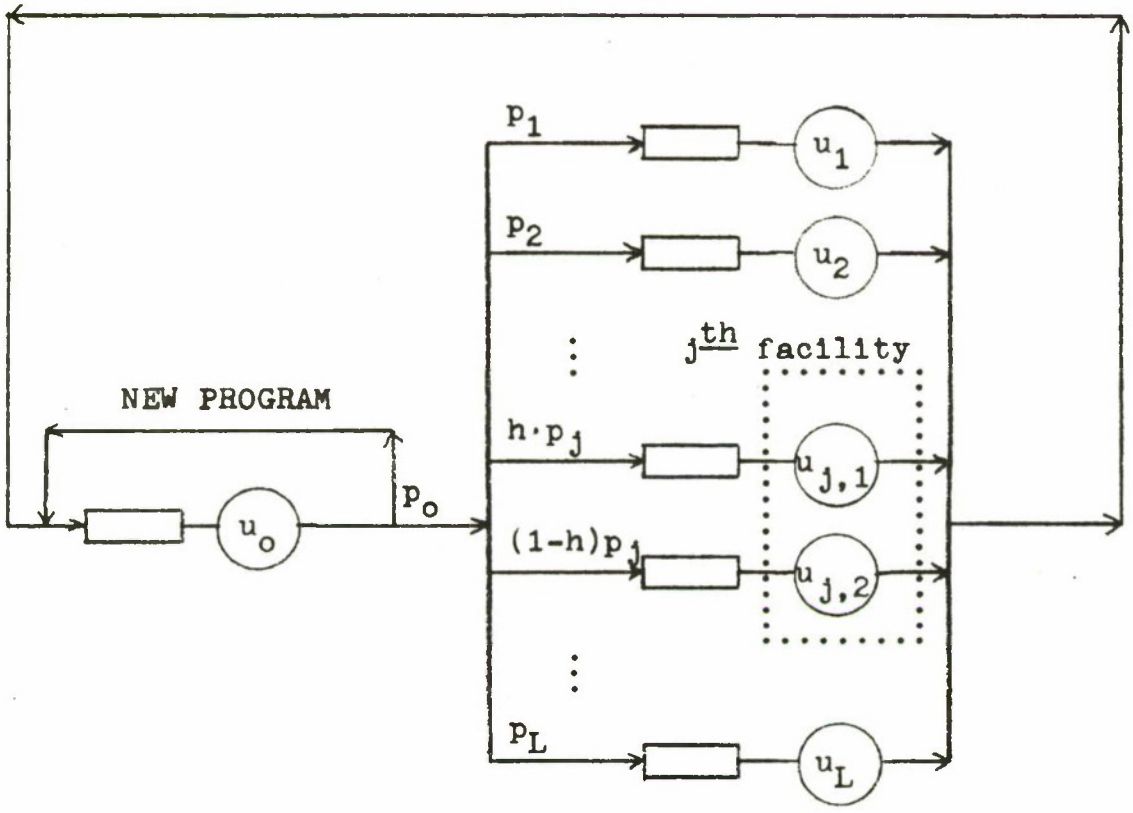


Figure 7-1

Hyperexponential Dedicated Peripheral Processors

state distribution for this network will have the general form given in equation 7-1 and can be immediately written down in any particular case. Since each state $(n_o, n_1, \ldots, n_{j,1}, n_{j,2}, \ldots, n_L)$ in the conceptually modified network maps into the state $(n_o, n_1, \ldots, n_{j,1}+n_{j,2}, \ldots, n_L)$ in the original network, the steady state distribution for the original network is given by

$$P(n_o, n_1, \ldots, n_j, \ldots, n_L) = \sum_{n_{j,1}=0}^{n_j} P(n_o, n_1, \ldots, n_{j,1}, n_j - n_{j,1}, \ldots n_L) \qquad 7\text{-}6$$

Note that the same solution technique can be used for hyper-exponential distributions of arbitrary degree.

To illustrate another method of constructing non-exponential service times, suppose that the $j^{\underline{th}}$ service facility in a central server network consists of a set of dedicated peripheral processors whose service times are the sum of k exponentially distributed random variables with means $1/u_{j,1}$, $1/u_{j,2}$, ..., $1/u_{j,k}$. The $j^{\underline{th}}$ service facility can then be conceptually divided into k individual exponential service facilities operating in series as illustrated in Figure 7-2. While the network in Figure 7-2 does not entirely conform to the specifications of the central server model, its steady state distribution can still be obtained rather easily using the methods of Appendix B.

Note first that the matrix P of branching probabilities has the following form:

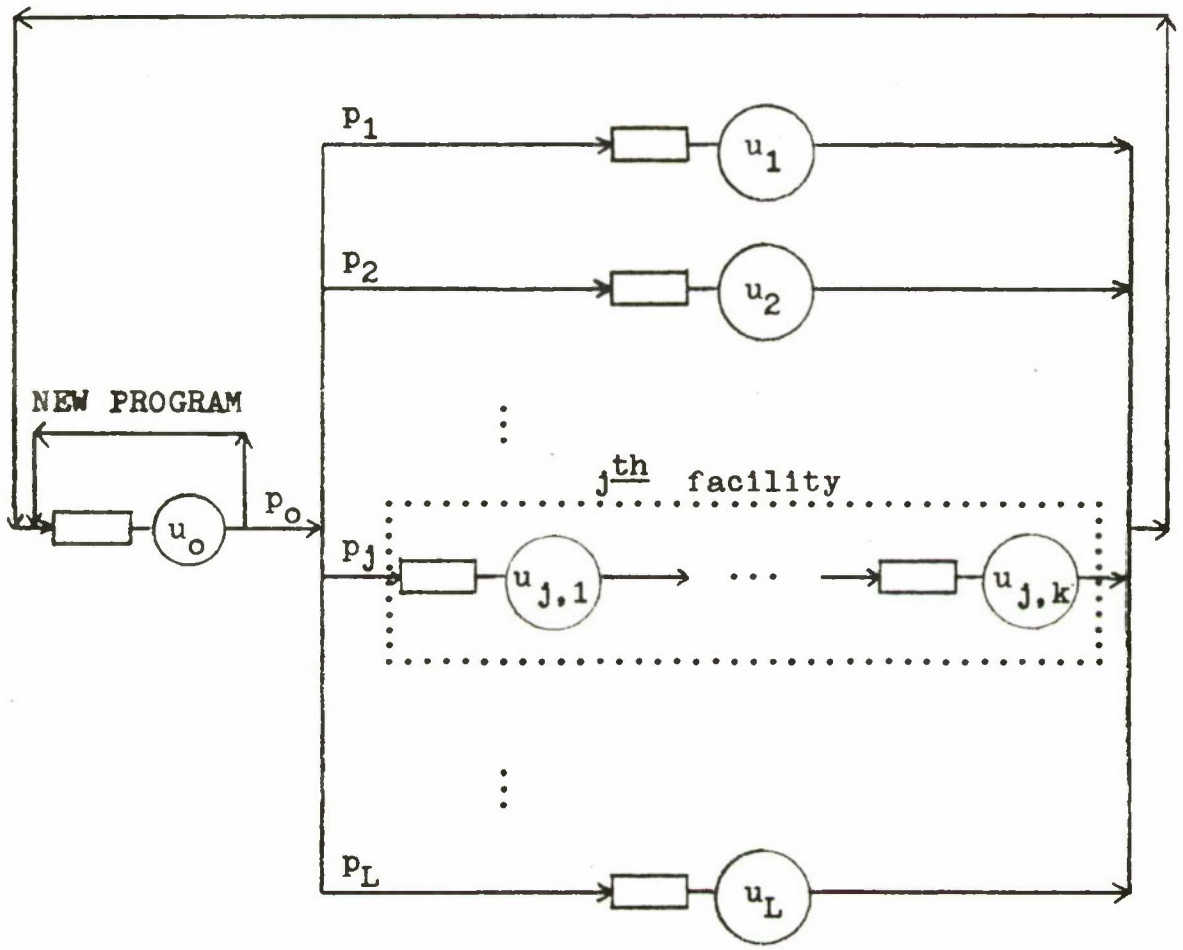


Figure 7-2

Erlang Sum Dedicated Peripheral Processors

					column j				column $j+k-1$			
	p_o	p_1	p_2	...	p_j	0	0	...	0	p_{j+1}	...	p_L
	1	0	0	...	0	0	0	...	0	0	...	0
	1	0	0	...	0	0	0	...	0	0	...	0
	⋮											
	1	0	0	...	0	0	0	...	0	0	...	0
row $j \rightarrow$	0	0	0	...	0	1	0	...	0	0	...	0
	0	0	0	...	0	0	1	...	0	0	...	0
	⋮											
row $j+k-2 \rightarrow$	0	0	0	...	0	0	0	...	1	0	...	0
row $j+k-1 \rightarrow$	1	0	0	...	0	0	0	...	0	0	...	0
	⋮											
	1	0	0	...	0	0	0	...	0	0	...	0

The equation $\underline{y} = \underline{y}P$ then becomes

$$y_o = p_o y_o + y_1 + y_2 + \cdots + y_{j-1} + y_{j,k} + y_{j+1} + \cdots + y_L$$

$$y_1 = p_1 y_o$$

$$y_2 = p_2 y_o$$

$$\vdots$$

$$y_{j-1} = p_{j-1} y_o$$

$$y_{j,1} = p_j y_o$$

$$y_{j,2} = y_{j,1}$$

$$y_{j,3} = y_{j,2}$$

$$\vdots$$

$$y_{j,k} = y_{j,k-1}$$

$$y_{j+1} = p_{j+1} y_o$$

$$\vdots$$

$$y_L = p_L y_o$$

It is thus clear that the vector

$$\underline{y} = (y_o, p_1 y_o, p_2 y_o, \ldots, p_{j-1} y_o, \underbrace{p_j y_o, \ldots, p_j y_o}_{\text{k components}}, p_{j+1} y_o, \ldots, p_L y_o)$$

satisfies the equation $\underline{y} = \underline{y}P$ for any value of y_o, and in particular for $y_o = u_o$. The steady state distribution for the conceptually modified network can then be immediately

obtained from equation B-15. The steady state distribution for the original network is then

$$P(n_o,n_1,\ldots,n_j,\ldots,n_L) = \sum_{\substack{k \\ \sum_{i=1} n_{j,i}=n_j}} P(n_o,\ldots,n_{j,1},\ldots,n_{j,k},\ldots,n_L) \qquad 7\text{-}7$$

The method used to obtain equations 7-6 and 7-7 can obviously be extended to include arbitrary parallel and series combinations of exponential components, although the solution of the equation $\underline{y} = \underline{y}P$ may then no longer be routine. Unfortunately there appears to be no easy way to extend this technique to shared (i.e., non-dedicated) service facilities of the type used in the original central server model. The problem is that such service facilities create additional queueing delays which make it difficult to characterize service times as simple combinations of parallel and series exponential delays. However it is still possible to write down the complete set of equilibrium equations and to attempt to solve them directly for specific cases.

Hyperexponential Central Processors with Processor Sharing*

The notion of processor sharing was discussed at some length in the section of Chapter 2 dealing with quantum controlled service disciplines. In this section processor

*The material presented in this section was suggested by F. Baskett of the University of Texas.

sharing will simply be regarded as the limit of a round robin service discipline in which the quantum size has shrunk to zero. Thus, if there are n_o programs present at the CPU, each will receive $1/n_o$ of the CPU's processing capacity.

Assume next that the amount of (full capacity) service time per processing request is a random variable with hyper-exponential density function of the form

$$h \cdot u_{o,1} \cdot e^{-u_{o,1}t} + (1-h) \cdot u_{o,2} \cdot e^{-u_{o,2}t}$$

Applying the technique of Figure 7-1, the CPU may then be conceptually divided into a pair of parallel exponential service facilities as illustrated in Figure 7-3. However Figure 7-3 cannot be considered as an exact analog of Figure 7-1 because it is assumed in Figure 7-3 that there is only a single CPU, and that this CPU is operating under a processor sharing service discipline. Thus the assumption of a service facility composed of dedicated processors is not valid in this case.

Continuing with the analysis, suppose that there are $n_{o,1}$ programs present at the upper CPU service facility and $n_{o,2}$ programs present at the lower CPU service facility. Each program thus receives $1/(n_{o,1}+n_{o,2})$ of the CPU's total capacity. It then follows that the rate of departure from the upper facility is $\frac{n_{o,1}}{n_{o,1}+n_{o,2}} \cdot u_{o,1}$ and the rate of departure from the lower facility is $\frac{n_{o,2}}{n_{o,1}+n_{o,2}} \cdot u_{o,2}$.

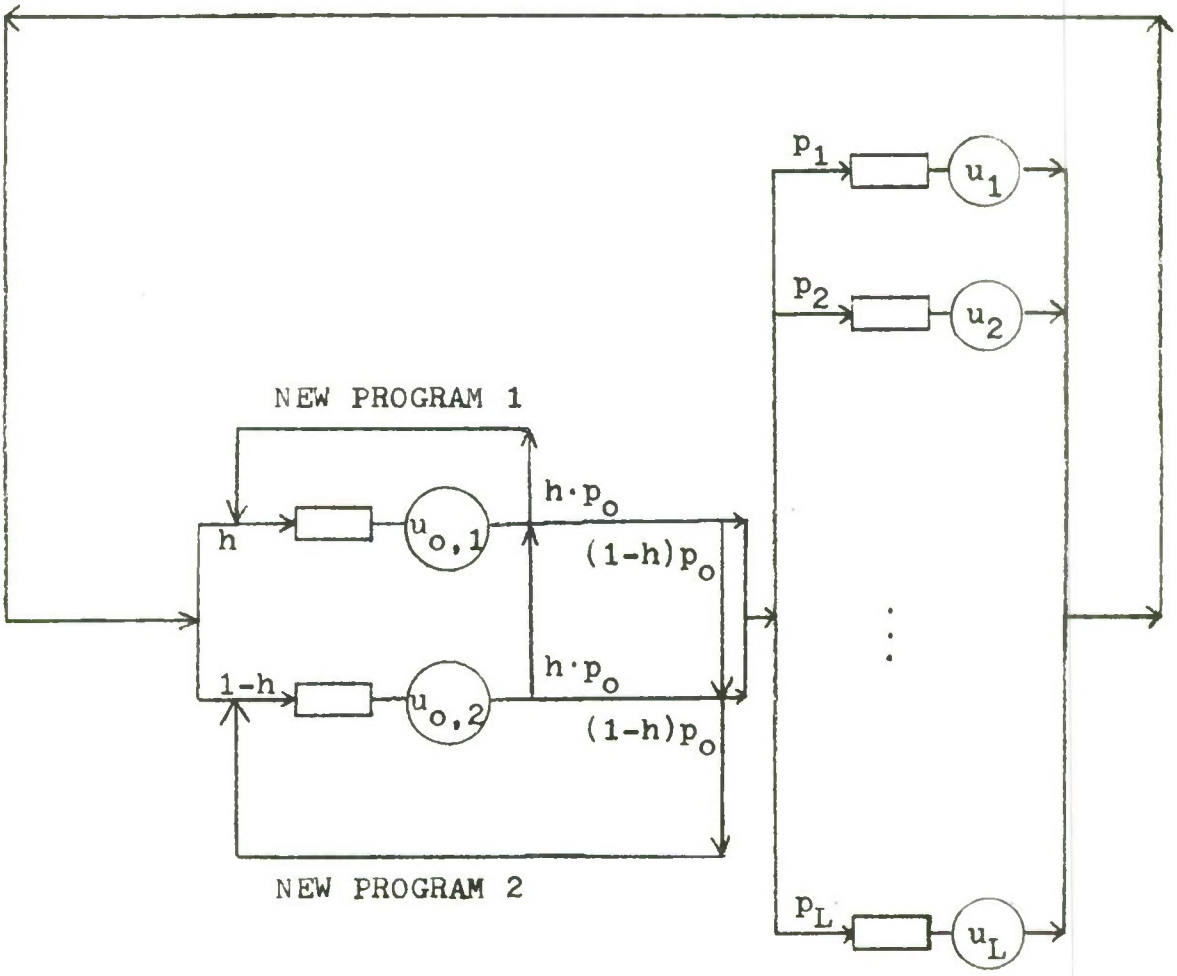


Figure 7-3

Hyperexponential CPU with Processor Sharing

Note that the processing rate of each service facility is not solely a function of the number of programs present at that facility. Thus it is not possible to use the solution techniques developed by Jackson and by Gordon and Newell in this case. It is of course still possible to write down the complete set of equilibrium equations and attempt to solve them directly. Baskett (8) has successfully carried out such an analysis for the case in which the only other component in the network is a set of dedicated exponential peripheral processors.* An extension of Baskett's work to the full central server model would be of considerable interest.

*This corresponds to the finite source Poisson arrivel process classified as type M_f in Chapter 2.

VARIATIONS IN THE DEGREE OF MULTIPROGRAMMING

A Specialized Time-Sharing Model

There are a number of ways in which the central server model can be extended to include random fluctuations in the degree of multiprogramming (i.e., the value of N). One possible approach is related to the observation that such fluctuations are almost always present in systems with interactive time-sharing terminals since programs which are waiting for responses from these terminals are not normally maintained in main memory. Thus the true level of multiprogramming in such systems is equal to N minus the number of programs waiting for terminal I/O.

Interactive time-sharing terminals have already been discussed in this chapter in the section dealing with dedicated peripheral processors. However it is necessary to extend the central server model still further to explicitly represent the fact that programs lose and then regain their main memory allocation as they go into and out of terminal wait states.

One simple way of representing this phenomenon is to assume that the system contains a PPU which will be called an overlay processor. Basically an overlay processor saves regions of main memory on auxiliary storage and then loads program and data segments into these regions. It is assumed that the overlay processor functions during the normal course

of program operation as well as at times when programs attempt to regain their main memory allocation after a terminal wait. Thus the overlay processor is similar in many respects to a page replacement processor.

Figure 7-4 represents a system containing an overlay processor and a set of interactive time-sharing terminals. Note that programs completing terminal I/O must obtain service from the overlay processor before proceeding to the CPU queue. In addition programs make requests for service from the overlay processor during their normal course of operation with probability p_{L-1}.

The steady state distribution for the network in Figure 7-4 can be readily obtained using the solution technique discussed in Appendix B. Note first that the matrix P of branching probabilities has the following form:

$$P = \begin{bmatrix} 0 & p_1 & p_2 & \cdots & p_{L-1} & p_L \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & & & & & \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

The equation $\underline{y} = \underline{y}P$ then becomes

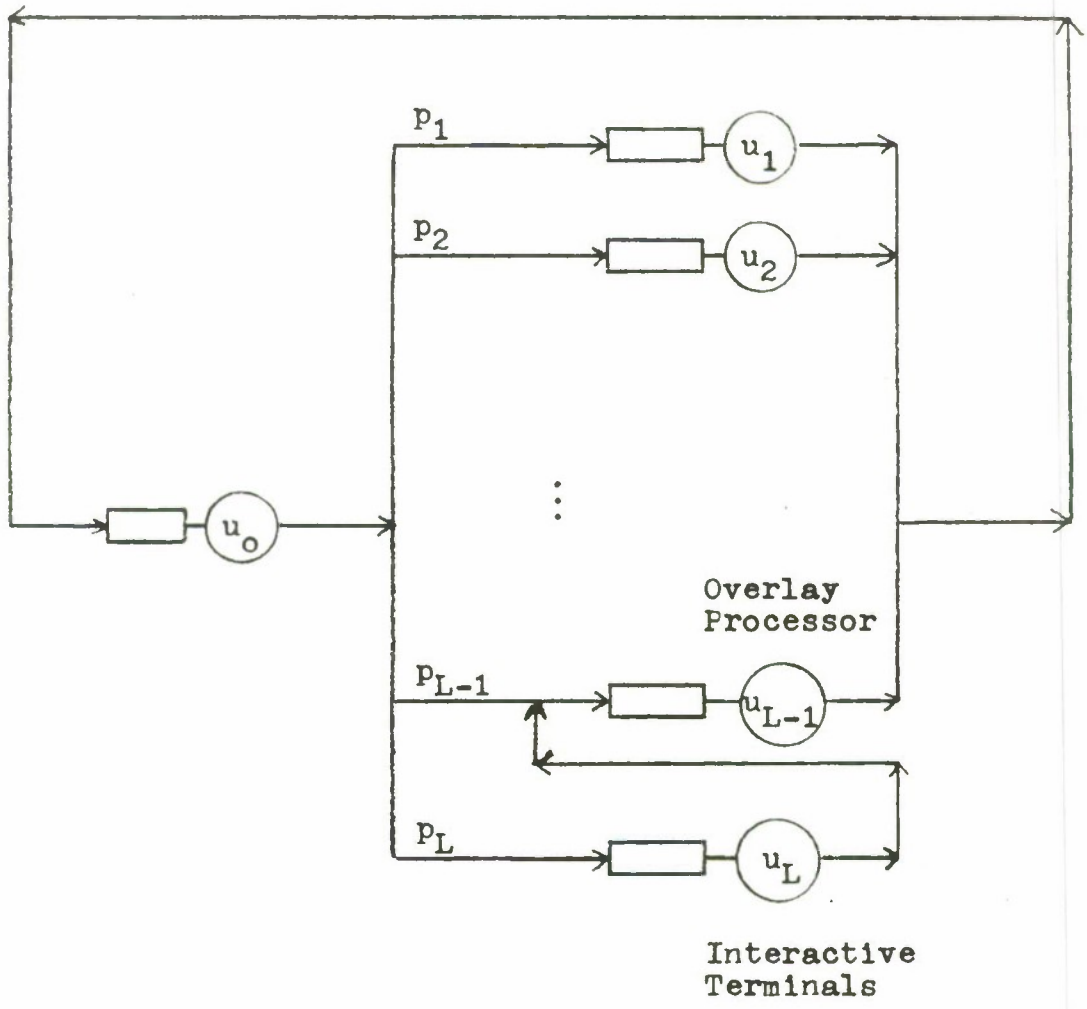


Figure 7-4

Specialized Time-Sharing Model

$$y_o = y_1 + y_2 + \ldots + y_{L-1}$$

$$y_1 = p_1 y_o$$

$$y_2 = p_2 y_o$$

$$\vdots$$

$$y_{L-1} = p_{L-1} y_o + p_L$$

$$y_L = p_L y_o$$

Thus the vector $\underline{y} = (y_o, p_1 y_o, p_2 y_o, \ldots, (p_{L+1} + p_L) y_o, p_L y_o)$ satisfies the equation $\underline{y} = \underline{y} P$ for any value of y_o, and in particular for $y_o = u_o$. The steady state distribution for the network can then be immediately obtained from equation B-15.

Note that Figure 7-4 contains no NEW PROGRAM path. This is to emphasize the fact that, while the original central server model is essentially a batch processing model, the model in Figure 7-4 is more properly regarded as a time-sharing model. Thus it is no longer sufficient to use the number of programs processed per unit time as the sole measure of system performance. Instead it is necessary to introduce measures which take response time and total number of active terminals into account. One possible measure is the maximum number of active terminals which can be supported at a given level of responsiveness. However this measure may not prove satisfactory for all applications, and so

additional work may be necessary before the model in Figure 7-4 can be used in conjunction with various optimization procedures.

An Open Network Model

An entirely different approach to the problem of fluctuations in the degree of multiprogramming is to assume that new programs arrive at the system in a random fashion from an unspecified external source and that programs disappear entirely from the system after they have completed their processing requirements.

Figure 7-5 illustrates such a system. It is assumed that the external arrivals are generated by a Poisson arrival process with mean rate $u_x \cdot a_x(N)$ where a_x is an arbitrary non-negative function of N, the total number of programs in the system at any given time. In addition it is assumed that the probability that a program will exit from the system after completing a CPU processing request is equal to p_o. Note that Figure 7-5 can be regarded as a standard central server network in which the NEW PROGRAM path has been cut open to permit external arrivals and departures.

Since Figure 7-5 is not a closed network its steady state distribution cannot be obtained using the method of Gordon and Newell. However Jackson's more general solution technique is clearly applicable. First note that the matrix P of branching probabilities has the following form:

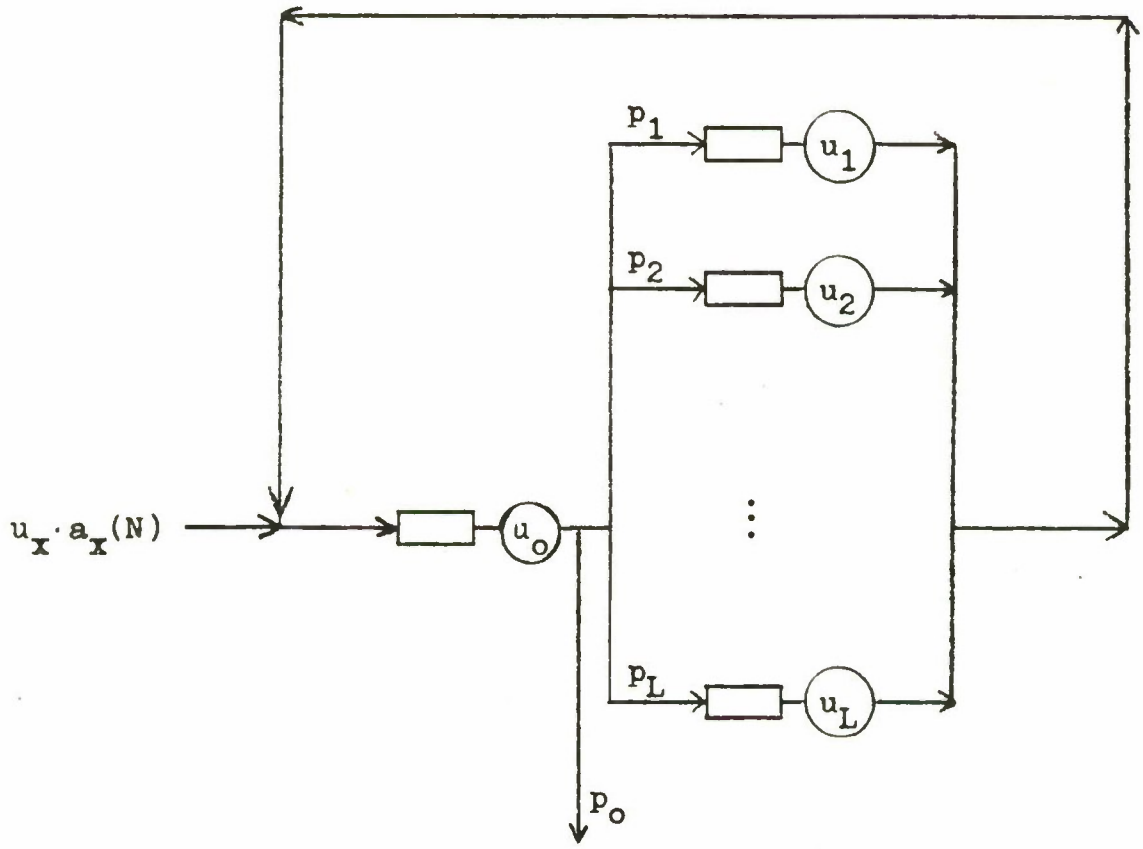


Figure 7-5

Open Network Model

$$P = \begin{bmatrix} 0 & p_1 & p_2 & \cdots & p_L \\ 1 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

Next note that arriving programs all proceed with probability one to the central server. Thus the vector which characterizes the branching probabilities for programs arriving from external sources is

$$\underline{e} = (1,0,0,\ldots,0)$$

To obtain the steady state distribution for the network it is necessary to solve the equation $\underline{y} = \underline{e} + \underline{y} \cdot P$ for the vector $\underline{y}$. Broken down into individual components, this equation is

$$y_o = 1 + y_1 + y_2 + \ldots + y_L$$

$$y_1 = p_1 y_o$$

$$y_2 = p_2 y_o$$

$$\vdots$$

$$y_L = p_L y_o$$

Thus the solution vector is

$$\underline{y} = (1/p_o,\ p_1/p_o,\ p_2/p_o,\ \ldots,\ p_L/p_o)$$

It is now necessary to define auxiliary functions similar to those defined in Appendix B.

$$\text{Let } A_x(n) = \begin{cases} 1 & \text{if } n = 0 \\ \prod_{k=1}^{n} a_x(k) & \text{if } n > 0 \end{cases}$$

Assume that the values of $A_j(n)$ for $j=0,1,2,\ldots,L$ are defined as in equation B-11. Finally let

$$N = n_o + n_1 + n_2 + \ldots + n_L$$

It then follows from the work of Jackson (41, p. 138) that the steady state distribution for the network in Figure 7-5 is given by

$$P(n_o,n_1,\ldots,n_L) = \frac{1}{G} (u_x)^N A_x(N) \frac{(1/u_o p_o)^{n_o}}{A_o(n_o)} \prod_{j=1}^{L} \frac{(p_j/u_j p_o)^{n_j}}{A_j(n_j)}$$

where the normalizing constant G is defined as

$$G = \sum_{N=0}^{\infty} \left[(u_x)^N A_x(N) \sum_{\sum_{j=0}^{L} n_j = N} \frac{(1/u_o p_o)^{n_o}}{A_o(n_o)} \prod_{j=1}^{L} \frac{(p_j/u_j p_o)^{n_j}}{A_j(n_j)} \right]$$

Since the total amount of main memory is limited, it becomes increasingly unlikely that new programs will be admitted to the system as the number of programs already in the system grows larger. The values of $a_x(N)$ can be speci-

fied in a way that reflects this fact. If there exists some upper bound on N beyond which no programs can be admitted (i.e., $a_x(N) = 0$ for $N > M$), then the system can be converted to a closed system with M circulating customers as indicated in Figure 7-6. The processing rate of the $x^{\underline{th}}$ server in Figure 7-6 is assumed to be $u_x \cdot a_x(M-n_x)$ where n_x is the number of customers present at the $x^{\underline{th}}$ server and a_x is the original external arrival rate function. Note that the value of $M-n_x$ in Figure 7-6 corresponds to the value of N in Figure 7-5.

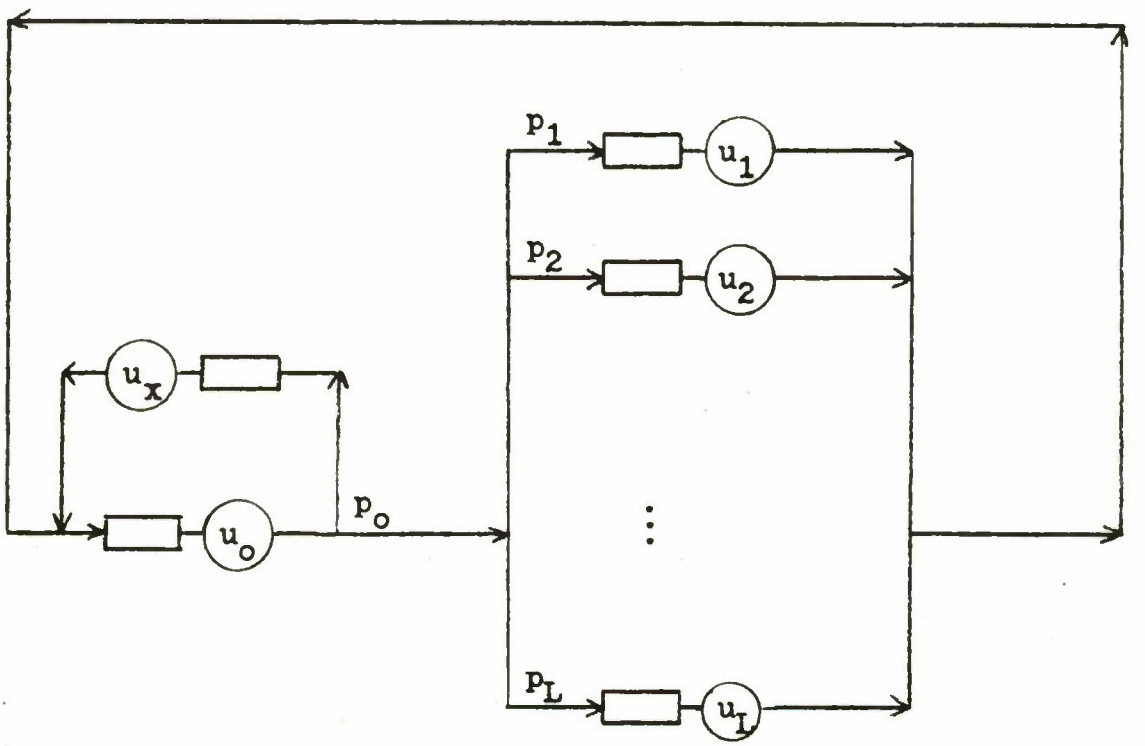


Figure 7-6

Equivalent Closed Network Model

CHAPTER 8: THE MODEL IN PERSPECTIVE

RELATION TO OTHER WORK

Introduction

The analytic network models discussed in the final section of Chapter 2 bear very little resemblance to the central server model presented in this thesis. However Smith's (76) numerical queueing network model does fall within the central server framework. In addition, both Arden and Boettner (6) and Fenichel and Grossman (33) have discussed certain non-queueing theoretic aspects of central server networks. Thus, despite the almost total lack of analytic studies of complex queueing network models, the general schematic framework of the central server model is not entirely without precedent.

In light of these remarks it is interesting and somewhat surprising to note that four completely independent analyses of the central server model have been published in the past few months. The first of these is contained in a Japanese language article by Tanaka (79) which appeared in October 1970. Analyses by Arora and Gallo (7) and Buzen (12) then appeared concurrently during the first week of April 1971 in conjunction with the SIGOPS Workshop of System Performance Evaluation.* Finally, a Ph.D dissertation deal-

*This Workshop was held at Harvard University on April 5-7, 1971. Proceedings may be obtained through the ACM.

ing with the central server model was presented by Moore (62) later in April 1971. It should be noted that Moore also discussed his model during the 38th National Operations Research Society of America Meeting held in October 1970. However no conference proceedings were published.

The relationship between the work presented in this thesis and the work of Tanaka, Arora and Gallo, and Moore will now be examined on an individual basis. Following this a brief account of the material that is unique to this thesis alone will be presented.

The Work of C.G. Moore

The original motivation for Moore's model was provided by the University of Michigan Terminal System (MTS). Since MTS is primarily a time-sharing system the model includes dedicated interactive terminals of the type discussed in Chapter 7.

Moore's derivation of the steady state distribution is based on the work of Gordon and Newell (41). After obtaining the steady state distribution Moore uses the results of a series of MTS measurements to assign numerical values to model parameters. The model is then used to make behavior predictions for MTS. Moore found a reasonable level of correlation between predicted bahavior and actual behavior, thus validating the model for this particular case.

From the point of view of this dissertation the most significant aspect of Moore's work is that it demonstrates that the assumptions which underlie the central server model are sufficiently realistic to permit the model to be of practical value in predicting the performance of actual multiprogramming systems.

The Work of S.R. Arora and A. Gallo

Arora and Gallo's development of the central server model grew out of consideration of an airline reservation system. As a result the peripheral servers in their model are identified with levels within a memory hierarchy rather than more general peripheral processors. After presenting the model in this somewhat specialized context, Arora and Gallo proceed to derive the steady state distribution without utilizing the results of Jackson (48) or Gordon and Newell (41). They then use the performance predictions of the model to evaluate alternative system configurations, using actual cost figures and functional characteristics to characterize the hardware and using empirically obtained program behavior statistics to characterize the processing load.

In a separate section of their paper Arora and Gallo consider the problem of optimal loading of program and data segments into the levels of the memory hierarchy under the assumption that the size of the various program and data

segments and the capacity of the levels in the hierarchy are given as parameters. Since the loading strategy determines the relative frequency with which the levels in the hierarchy will be accessed, this problem is related to the peripheral processor utilization problem analyzed in Chapter 6. However the objective function that Arora and Gallo attempt to optimize does not take into account the queueing delays in the system. Instead it is simply a linear function of the access times and transfer rates of the non-executable memory levels plus the cycle times of the executable memory levels. The specialized form of the objective function reduces the optimization problem to a problem in linear programming which is then solved by Vogel's method.

Arora and Gallo's choice of objective function is somewhat surprising in light of the fact that the overall measure of performance they use to evaluate alternative system configurations does indeed take queueing delays into account. Thus in the extreme case where the fastest level of non-executable memory has virtually unlimited capacity, Arora and Gallo's optimal loading strategy will place all non-directly executable program and data segments into that level even though this will almost certainly result in excessive queueing delays and a sub-optimal level of performance (assuming that the level of performance is determined by the measure developed in the other part of the paper). This difficulty is avoided in the analysis of Chapter 6 since

there the objective function and the measure of system performance are one in the same.

The Work of H. Tanaka

Tanaka's model is similar to Moore's in that it is oriented towards time-sharing systems. Hence this model also includes dedicated interactive terminals of the type discussed in Chapter 7. The major goal of this paper is the derivation of the steady state distribution and related expressions such as queueing delays and overall response time. No attempt is made to apply the model to theoretical problems of the type presented in Chapter 6 or to validate the model by consideration of empirical data.

Tanaka's derivation of the steady state distribution for central server networks having an arbitrary number of parallel servers at each service facility was carried out from first principles without utilizing the results of Jackson or Gordon and Newell. This represents a significant accomplishment even though it is possible to derive this distribution in a simpler manner by making use of these related results.

New Material

This thesis treats a number of topics which were not considered by the previous authors. For example the Conservation laws (i.e., equations 5-6 and 5-8) and the fact

that the the most highly saturated server has the longest expected queue (which follows from equation 5-10) represent new results. In addition the three theoretical problems treated in Chapter 6 have not been analyzed elsewhere, although Arora and Gallo have considered the problem of optimal peripheral processor utilization in a different context.

Certain technical points such as the use of the NEW PROGRAM path to represent program terminations are also new. Finally, the computational algorithms presented in the second half of Chapter 5 are new and should be of considerable value in the analysis of any queueing network model whose steady state distribution can be derived using the methods of Jackson and Gordon and Newell. The significance of these algorithms thus extends well beyond the scope of the central server model itself.

SUGGESTIONS FOR FURTHER RESEARCH

A number of the earlier chapters of this thesis contain explicit references to promising areas for future research. For example, several problems associated with quantum controlled service disciplines are discussed on pages 34 - 36 of Chapter 2, the problem of limited queue size with induced blocking is mentioned on page 75 of Chapter 3, and the need for additional research on the problem of page traffic balancing is cited on page 177 of Chapter 6. In addition, the extensions to the basic model discussed in Chapter 7 can be used to construct a host of models which closely resemble particular systems of interest.

With regard to this last point it should be noted that the construction of models of particular systems or classes of systems does not in and of itself constitute a research activity. For example, the construction of a mathematical model for the purpose of predicting the behavior of an actual or proposed system generally falls under the heading of engineering. This is especially true if the mathematical techniques used to construct the model are highly standardized and if the performance predictions are being used to guide system design. Since the purpose of this section is to discuss prospective research problems, such engineering activities will not be considered further.

Research activities are primarily concerned with investigating the underlying factors which influence the behavior of all systems of a given type. A number of interesting research problems can be formulated within the framework of the central server model. One, which in some ways resembles the buffer size determination problem of Chapter 6, will be called the program organization problem. The problem is simple to state. Assume that the total net amount of processing time per program is specified for each processor in a system. Then, taking overhead and the effects of buffer size on degree of multiprogramming into account, determine how this total processing load should be organized in order to optimize system performance. In other words, specify the values of the model parameters ($u_0,\ldots,u_L,p_0,\ldots,p_L$ and N) which optimize overall performance, subject to the constraint that the net amount of processing time per program for each processor is constant. The solution to this problem may provide valuable insight into the relationship between program organization and system architecture under a variety of processing loads.

A somewhat different problem has its initial motivation in real-time system design. Suppose a particular routine is executed periodically in response to an external interrupt, and assume that the time constraints associated with the interrupt are sufficiently lax so that the routine can be maintained in secondary storage if desired. Then, given the

average time between interrupts, the size of the routine, the time required to access it from secondary storage, and the utilization factor for the secondary storage access channel, specify the conditions under which it is preferable for the routine to reside in main memory and the conditions under which it is preferable for the routine to reside in secondary storage. This problem, which will be called the residency problem, has implications for the management of monitor segments and utility routines in conventional multiprogramming systems in addition to its original application to real-time system design.

Still another problem is that of optimizing system performance by altering the external processing load (i.e., the job mix) in various ways. For example, it is possible to alter the external processing load by changing the relative percentages of compute bound and I/O bound jobs, or by adding a real-time job stream with certain processing characteristics. Since such modifications can affect the average amount of processing per program, it may no longer be possible to compare systems on the basis of number of programs processed per unit time. Consequently, solutions to problems of this type may require the development of new measures of system performance.

On a more theoretical level, the effects of different processing time distributions, and especially the effect of changes in the variance of these distributions, should prove

interesting to explore. The work of Baskett (8) which is discussed on page 192 of Chapter 7 appears to be a promising start in this direction.

In short, the prospects for future research in the area of queueing network models in general and the central server model in particular appear quite promising. It thus seems likely that the central server model and its variants will become the objects of extensive examination in the coming years.

APPENDIX A: THE EXPONENTIAL DISTRIBUTION

From a mathematical standpoint, an exponentially distributed random variable is one whose probability density function is ae^{-at} for some $a > 0$.* Given this definition it is possible to derive a number of formal properties which exponentially distributed random variables satisfy. However, to gain real insight into the nature of this distribution it is often more helpful to regard it as the limiting case of an intuitively simpler discrete time process.

It is useful to have a specific example in mind when considering the exponential distribution in this light. Suppose then that in a queueing system each customer presents the server with the following request: namely, to toss a particular coin once every s seconds until a "head" appears. As soon as the first "head" is reached, the request is considered satisfied and the customer departs.

Suppose the the coin is unbalanced so that the probability of getting a "head" on any particular toss is h $(0 < h < 1)$. Then the probability that the server will require s seconds (i.e., one toss) to complete a customer's request is equal to h, the probability that the server will require exactly 2s seconds (i.e., two tosses) to complete a customer's request is equal to (1-h)h, and in general the

---

*Equivalently, an exponentially distributed random variable may be defined as a random variable whose cumulative probability distribution function is $1-e^{-at}$ for some $a > 0$.

probability that the server will require exactly ns seconds to complete a customer's request is equal to $(1-h)^{n-1}h$. Note that these formulas are based on the assumption that the outcome of any particular coin toss is independent of all other coin tosses. That is, each toss is an independent Bernoulli trial with probability of success equal to h.

This assumption of independence has a number of interesting consequences. First of all, it implies that regardless of the amount of service a customer has already received, the probability that his service request will be completed on the next coin toss is always equal to h. More generally, if a customer receiving service is observed at an arbitrary point in time, the probability that his service request will be completed on the $n^{\underline{th}}$ coin toss after that point in time is equal to $(1-h)^{n-1}h$. This is true regardless of the amount of service the customer had already received before he was observed. Thus, the amount of service a customer has already received in no way affects the probabilities governing the the additional service he can expect to receive. Probability distributions satisfying this condition are known as memoryless distributions. The particular memoryless distribution cited in this example is known as the geometric distribution.

When service times are geometrically distributed, a customer's service time is always some integral multiple of s, the basic coin tossing interval. However, in most situations of interest service times range over the entire

continuum of positive values. It would thus be useful to define a service time distribution which ranges over this continuum and which also satisfies the memoryless property.

One way such a distribution might be constructed is by starting with the original example and then letting s, the interval between tosses, approach zero. This operation introduces certain complications since the expected number of coin tosses required to complete a customer's service is $\sum_{n=1}^{\infty} n(1-h)^{n-1}h = 1/h$, and so the expected amount of time required to complete a customer's service request is s/h seconds. Thus, if s is allowed to approach zero, the expected amount of time required to complete a customer's service will also approach zero, and in the limit each customer is served in zero time.

This difficulty may be avoided if h is also required to approach zero as s does. In particular, if the ratio s/h is held constant as s approaches zero, the expected amount of time to serve a customer will remain constant even though service times will, in the limit, range over the entire continuum of positive real values. This limiting process may be envisaged as one in which coin tosses become more and more frequent while the probability of getting a "head" on any particular toss becomes progressively less likely.

To complete this discussion it is necessary to determine the distribution of service times in this limiting case. Suppose that the ratio s/h is kept equal to 1/a for

some value of $a > 0$. Now, for any value of t, the probability that a customer will require more than t seconds to complete his service request is equal to the probability that he will require at least t/s tosses. If $\boxed{t/s}$ is defined as the largest integer less than or equal to t/s, then this probability can be expressed as:

$$\sum_{n=\boxed{t/s}+1}^{\infty} (1-h)^{n-1}h = (1-h)^{\boxed{t/s}} = (1-as)^{\boxed{t/s}} \quad \text{since } s/h = 1/a .$$

Thus, in the limit, the probability that a customer will require more than t seconds of service is:

$$\lim_{s\to o} (1-as)^{\boxed{t/s}} = \lim_{s\to o} (1-as)^{t/s} \; * = \left[\lim_{s\to o} (1-as)^{1/s}\right]^{t} = (e^{-a})^{t} = e^{-at}$$

Therefore, the probability that a customer's service time is less than or equal to t is $1-e^{-at}$, which is to say that service times are exponentially distributed.

It is straightforward to verify that if service times are exponentially distributed, then the amount of service a

---

*The equality of the two limits derives from the fact that $(1-as)^{\boxed{t/s}} - (1-as)^{t/s}$ is bounded by as and hence can be made arbitrarily small. To see this, note that:

$$(1-as)^{\boxed{t/s}} - (1-as)^{t/s} < (1-as)^{\boxed{t/s}} - (1-as)^{\boxed{t/s}+1} = as(1-as)^{\boxed{t/s}} \leq as$$

from the power series expansions whenever $o < s < 1/a$

customer has already received in no way affects the probabilities governing the additional service he can expect to receive.* This fact should also be obvious from the preceding discussion since, intuitively speaking, at each point in time the server may be thought of as making a decision as to whether to eject the customer or to continue serving him until the next point in time, and the probability that the server will decide to eject the customer at any particular point in time is constant, independent of the amount of service the customer has already received. The value of understanding the exponential distribution on this admittedly vague and intuitive level is that the nature of the memoryless property, which is so crucial in queueing theory, becomes immediately apparent.

As a final point, it is worth noting that the geometric distribution is the only discrete distribution to satisfy the memoryless property and that the exponential distribution is the only continuous distribution to do so. Feller (32) demonstrates these facts in Sections XIII.9 (p. 328) and XVII.6 (p. 458) respectively.

---

*The probability that a customer will receive an additional v seconds or less of service, given that he has already received u seconds of service, is:

$$\frac{\int_u^{u+v} ae^{-at}\,dt}{\int_u^{\infty} ae^{-at}\,dt} = \frac{e^{-au} - e^{-a(u+v)}}{e^{-au}} = 1-e^{-av} = \int_0^v ae^{-at}\,dt$$

which is the probability that a customer just beginning service will receive a total of v seconds or less of service.

APPENDIX B: A SOLUTION TECHNIQUE FOR MARKOVIAN QUEUEING NETWORKS

Simple Exponential Servers

The purpose of this Appendix is to review the solution techniques used by Jackson (48) and Gordon and Newell (41) to obtain steady state distributions for certain classes of queueing networks. This first section illustrates the way in which steady state distributions can be obtained for networks made up of simple exponential servers. The solution technique is then extended to include queue dependent exponential servers in the second section of the Appendix.

This analysis treats closed queueing networks only. That is, it is assumed that a fixed number of customers circulate through the network at all times with no possibility of customers either entering or leaving. Such networks will be characterized as follows:

$L+1$ = the number of servers in the network.

u_j = the processing rate of the $j^{\underline{th}}$ server for $j=0,1,\ldots,L$ (i.e., the service time at the $j^{\underline{th}}$ server is an exponentially distributed random variable with mean $1/u_j$).

p_{ij} = the probability that a customer leaving the $i^{\underline{th}}$ server will proceed to the $j^{\underline{th}}$ server. Clearly $\sum_{j=0}^{L} p_{ij} = 1$ for $i=0,1,\ldots,L$.

N = the number of customers circulating in the network.

Assume that it is desired to obtain

$P(n_o,n_1,\ldots,n_L)$ = the steady state probability that there are n_j customers present at the $j^{\underline{th}}$ server. Note that these probabilities are only defined for cases in which $\sum_{j=0}^{L} n_j = N$ and $0 \leq n_j \leq N$.

Before determining the steady state probabilities it is useful to define one auxiliary function. Let

$$e(n_j) = \begin{cases} 0 & \text{if } n_j = 0 \\ 1 & \text{if } n_j > 0 \end{cases}$$

It is now possible to begin the analysis. Note first that the rate of transition out of state $(n_o,n_1,\ldots,n_L)$ at equilibrium is

$$\sum_{j=0}^{L} e(n_j)\ u_j\ P(n_o,n_1,\ldots,n_L)$$

This formula expresses the fact that customers exit from state $(n_o,n_1,\ldots,n_L)$ through the $j^{\underline{th}}$ server as long as there is at least one customer present at that server. If no customers are present at the $j^{\underline{th}}$ server, no transitions are possible and the factor $e(n_j)$ will set the corresponding term in the summation equal to zero.

Next note that the rate of transition into state $(n_o,n_1,\ldots,n_L)$ at equilibrium is

$$\sum_{i=0}^{L} \sum_{j=0}^{L} e(n_j)\, u_i\, p_{ij}\, P(n_o,n_1,\ldots,n_i+1,\ldots,n_j-1,\ldots,n_L)$$

This formula expresses the fact that transitions occur into state $(n_o,n_1,\ldots,n_L)$ from state $(n_o,n_1,\ldots,n_i+1,\ldots,n_j-1,\ldots,n_L)$ whenever a customer completes service at the $i^{\underline{th}}$ server and then proceeds to the $j^{\underline{th}}$ server. Since the $i^{\underline{th}}$ server operates at rate u_i and transitions from the $i^{\underline{th}}$ server to the $j^{\underline{th}}$ server occur with probability p_{ij}, the rate of transition into state $(n_o,n_1,\ldots,n_L)$ from state $(n_o,n_1,\ldots,n_i+1,\ldots,n_j-1,\ldots,n_L)$ is equal to $u_i\, p_{ij}\, P(n_o,n_1,\ldots,n_i+1,\ldots,n_j-1,\ldots,n_L)$.

This transition rate must be multiplied by $e(n_j)$ to account for the fact that no such transitions can occur when $n_j=0$ since state $(n_o,n_1,\ldots,n_i+1,\ldots,n_j-1,\ldots,n_L)$ cannot exist in this case. Multiplication by $e(n_j)$ is necessary because, even though state $(n_o,n_1,\ldots,n_i+1,\ldots,n_j-1,\ldots,n_L)$ may not logically exist, a formal value of the function $P(n_o,n_1,\ldots,n_i+1,\ldots,n_j-1,\ldots,n_L)$ will always exist*.

---

*The function $P(n_o,n_1,\ldots,n_L)$ is given in equation B-8 . From a logical standpoint this function is only defined for

Next note that when j=i the corresponding term in the summation is

$$e(n_i)\ u_i\ p_{ii}\ P(n_o,n_1,\ldots,n_i,\ldots,n_L)$$

This term represents transitions from state $(n_o,n_1,\ldots,n_L)$ to itself which occur as a result of customers completing service at the $i^{\underline{th}}$ server and then immediately returning to that server. The multiplication by $e(n_i)$ accounts for the fact that such transitions can only occur if there is at least one customer present at the $i^{\underline{th}}$ server.

Since the rate of transition out of any state is equal to the rate of transition into that state at equilibrium, the steady state probabilities must all satisfy the following equation:

$$\sum_{j=0}^{L} e(n_j)\ u_j\ P(n_o,n_1,\ldots,n_L) =$$

$$\sum_{i=0}^{L}\sum_{j=0}^{L} e(n_j)\ u_i\ p_{ij}\ P(n_o,n_1,\ldots,n_i+1,\ldots,n_j-1,\ldots,n_L) \qquad \text{B-1}$$

---

cases in which $0 \leq n_j \leq N$ and the n_j sum to N. However, to simplify the formal manipulations of this section it is assumed that this function is defined for all values of n_j.

A separation of variables technique can be used to obtain values of $P(n_o,n_1,\ldots,n_L)$ which staisfy B-1. Assume that

$$P(n_o,n_1,\ldots,n_L) = \frac{1}{G}\prod_{k=0}^{L}(X_k)^{n_k} \qquad \text{B-2}$$

where G is a constant that will be specified later and the X_k are functions of the network parameters. To determine the X_k note first that

$$P(n_o,n_1,\ldots,n_i+1,\ldots,n_j-1,\ldots,n_L) = \frac{X_i}{X_j}\frac{1}{G}\prod_{k=0}^{L}(X_k)^{n_k} \qquad \text{B-3}$$

Substituting in B-1 from B-2 and B-3,

$$\sum_{j=0}^{L} e(n_j)\, u_j \frac{1}{G}\prod_{k=0}^{L}(X_k)^{n_k} =$$

$$\sum_{i=0}^{L}\sum_{j=0}^{L} e(n_j)\, u_i\, p_{ij}\, \frac{X_i}{X_j}\frac{1}{G}\prod_{k=0}^{L}(X_k)^{n_k}$$

Dividing through by $\frac{1}{G}\prod_{k=0}^{L}(X_k)^{n_k}$ and moving the results to one side,

$$\sum_{j=0}^{L} e(n_j)\, u_j \;-\; \sum_{i=0}^{L}\sum_{j=0}^{L} e(n_j)\, u_i\, p_{ij}\, (X_i/X_j) \;=\; 0$$

Hence, $$\sum_{j=0}^{L} e(n_j)\left[u_j \;-\; \sum_{i=0}^{L} u_i\, p_{ij}\, (X_i/X_j)\right] \;=\; 0 \qquad \text{B-4}$$

If all the customers are present at the $k^{\underline{th}}$ server, then $e(n_k)$ will be equal to one and all the other $e(n_j)$ will be equal to zero. In order to satisfy equation B-4 in this case, it is thus necessary that

$$u_k \;-\; \sum_{i=0}^{L} u_i\, p_{ik}\, (X_i/X_k) \;=\; 0 \qquad \text{B-5}$$

Since it is possible for all the customers to be present at any server in the network, equation B-5 must be satisfied for $k=0,1,\ldots,L$. In addition, it is obvious that equation B-4 will be satisfied for any state $(n_o,n_1,\ldots,n_L)$ if equation B-5 is satisfied for $k=0,1,\ldots,L$. Thus equation B-5 represents a necessary and sufficient set of conditions for determining the X_j.

It is possible to rewrite equation B-5 in a simpler form. First define

$$y_k = u_k X_k \qquad \text{for } k=0,1,\ldots,L \qquad \text{B-6}$$

Equation B-5 then becomes

$$y_k = \sum_{i=0}^{L} y_i\, p_{ik}$$

Since this equality must hold for $k=0,1,\ldots,L$, the vector $\underline{y} = (y_0,y_1,\ldots,y_L)$ must satisfy the eigenvector equation

$$\underline{y} = \underline{y}\, P \qquad \text{B-7}$$

where P is the matrix (p_{ij}) .

Thus, assuming a real and non-negative solution to equation B-7 can be found, it follows from equations B-6 and B-2 that the steady state distribution of customers in the network is

$$P(n_o,n_1,\ldots,n_L) = \frac{1}{G} \prod_{k=0}^{L} (y_k/u_k)^{n_k} \qquad \text{B-8}$$

Note that no explicit reference has been made thus far to the number of customers in the network (i.e., N). In fact the only part of the solution which depends on N is the constant G. To express this dependency G will be written as G(N) for the remainder of this discussion.

The constant G(N) is selected so that the sum of all the $P(n_o,n_1,\ldots,n_L)$ will be equal to one. Since any value of $P(n_o,n_1,\ldots,n_L)$ for which $\sum_{j=0}^{L} n_j = N$ represents a possible state of the system, it follows that

$$G(N) = \sum_{\sum_{k=0}^{L} n_k = N} \prod_{k=0}^{L} (y_k/u_k)^{n_k} \qquad \text{B-9}$$

The derivation of equations B-8 and B-9 is essentially a restatement of Gordon and Newell's argument. However the network description was slightly simplified by the assumption that the processing rate of each server is independent of the number of customers present at that server. The next

section demonstrates the way in which the steady state distribution can be obtained when such dependencies are assumed to be present.

Queue Dependent Servers

The network description is the same as in the previous section except that the processing rate of each server is no longer a constant but is instead a function of the number of customers present at the server. That is, if there are k customers present at the j^{th} server, then the time until the next service completion is assumed to be an exponentially distributed random variable with mean $\frac{1}{a_j(k)\cdot u_j}$.

In the previous section it was in effect assumed that $a_j(k)=1$ for $j=0,1,\ldots,L$ and $k=1,2,\ldots,N$. It will now be assumed that the a_j are arbitrary functions subject only to the constraint that $a_j(k)>0$ for $j=0,1,\ldots,L$ and $k=1,2,\ldots,N$. The equation which corresponds to B-1 is then

$$\sum_{j=0}^{L} e(n_j)\, a_j(n_j)\, u_j\, P(n_o,n_1,\ldots,n_L) =$$

$$\sum_{i=0}^{L} \sum_{\substack{j=0 \\ j\neq i}}^{L} e(n_j)\, a_i(n_i+1)\, u_i p_{ij}\, P(n_o,n_1,\ldots,n_i+1,\ldots,n_j-1,\ldots,n_L)$$

$$+ \sum_{i=0}^{L} e(n_i)\, a_i(n_i)\, u_i\, p_{ii}\, P(n_o,n_1,\ldots,n_i,\ldots,n_L) \qquad \text{B-10}$$

A steady state distribution which satisfies equation B-10 can also be obtained by a separation of variables technique, but in order to do so it is first necessary to make a variable transformation. Begin by defining

$$A_j(0) = 1$$

$$A_j(n) = \prod_{k=1}^{n} a_j(k) \quad \text{for} \quad n=1,2,\ldots,N \qquad \text{B-11}$$

Then define

$$Q(n_o,n_1,\ldots,n_L) = P(n_o,n_1,\ldots,n_L) \prod_{j=0}^{L} A_j(n_j)$$

Note that

$$P(n_o,n_1,\ldots,n_L) = \frac{Q(n_o,n_1,\ldots,n_L)}{\prod_{j=0}^{L} A_j(n_j)} \qquad \text{B-12}$$

Also

$$P(n_o,n_1,\ldots,n_1+1,\ldots,n_j-1,\ldots,n_L)$$

$$= \frac{\frac{a_j(n_j)}{a_i(n_i+1)}\, Q(n_o,n_1,\ldots,n_1+1,\ldots,n_j-1,\ldots,n_L)}{\prod_{j=0}^{L} A_j(n_j)} \qquad \text{B-13}$$

Substituting in B-10 from B-12 and B-13,

$$\frac{\sum_{j=0}^{L} e(n_j)\, a_j(n_j)\, u_j\, Q(n_o,n_1,\ldots,n_L)}{\prod_{j=0}^{L} A_j(u_j)} =$$

$$\frac{\sum_{i=0}^{L}\sum_{j=0}^{L} e(n_j)\, a_j(n_j)\, u_i p_{ij}\, Q(n_o,n_1,\ldots,n_i+1,\ldots,n_j-1,\ldots,n_L)}{\prod_{j=0}^{L} A_j(u_j)}$$

B-14

Multiplying through by $\prod_{j=0}^{L} A_j(u_j)$ reduces equation B-14 to the same form as equation B-1 except that all the P's are replaced by Q's and $e(n_j)$ is replaced everywhere by $e(n_j)\cdot a_j(n_j)$. It is thus possible to proceed exactly as in the case of equation B-1 and derive

$$\sum_{j=0}^{L} e(n_j)\, a_j(n_j) \left[u_j - \sum_{i=0}^{L} u_i\, p_{ij}\, (X_i/X_j) \right] = 0$$

Since $a_j(N)>0$ by hypothesis, it is thus possible to deduce

$$u_k - \sum_{i=0}^{L} u_i\, p_{ij}\, (X_i/X_j) = 0$$

for k=0,1,...,L by the same argument that was used to deduce B-5 from B-4. It then follows that

$$Q(n_o,n_1,\ldots,n_L) = \frac{1}{G(N)} \prod_{k=0}^{L} (y_k/u_k)^{n_k}$$

where $\underline{y} = (y_o,y_1,\ldots,y_L)$

is the real and non-negative solution of the eignevector

equation $\underline{y} = \underline{y}\cdot P$.

Applying equation B-12, it then follows that

$$P(n_o,n_1,\ldots,n_L) = \frac{1}{G(N)} \prod_{k=0}^{L} \frac{(y_k/u_k)^{n_k}}{A_k(n_k)} \qquad \text{B-15}$$

The normalizing constant G(N) is clearly determined by the

equation $$G(N) = \sum_{\sum_{k=0}^{L} n_k = N} \prod_{k=0}^{L} \frac{(y_k/u_k)^{n_k}}{A_k(n_k)} \qquad \text{B-16}$$

Equations B-15 and B-16 represent a minor generalization of the results obtained by Gordon and Newell. Jackson's results, on the other hand, are considerably more general and include these equations as a special case.

BIBLIOGRAPHY

1 Abate, J. and Dubner, H. Optimizing the performance of a drum-like storage. IEEE Trans. on Comp., C-18, 11 (Nov. 1969), 992-997.

2 Abate, J., Dubner, H. and Weinberg, S.B. Queueing analysis of the IBM 2314 disk storage facility. JACM, 15, 4 (Oct. 1968), 577-589.

3 Adiri, I. A time-sharing queue with preemptive-resume priority discipline. Israel J. of Tech., 6, 5 (Nov. 1968),277-282.

4 Adiri, I. Computer time-sharing queues with priorities. JACM, 16, 4 (Oct. 1969), 631-645.

5 Adiri, I. and Avi-Itzhak, B. A time-sharing queue with a finite number of customers. JACM, 16, 2 (Apr. 1969), 313-323.

6 Arden, B. and Boettner, D. Measurement and performance of a multiprogramming system. ACM Symposium on Operating Systems Principles, ACM, N.Y., Oct. 1969, 130-146.

7 Arora, S.R. and Gallo, A. The optimal organization of multiprogrammed multi-level memory. ACM-SIGOPS Workshop on System Performance Evaluation, ACM, N.Y., Apr. 1971, 104-141.

8 Baskett, F. Mathematical Models of Multiprogrammed Computer Systems. Ph.D. Thesis, Univ. of Texas, Austin, Texas, Dec. 1970.

9 Boudreau P.E. and Kac, M. Analysis of a basic queuing problem arising in computer systems. IBM J. Res. and Development, 5, 2 (Apr. 1961), 132-140.

10 Bowdon, E.K., Sr. Priority assignment in a network of computers. Digest IEEE 1969 Comp. Group Conf., IEEE Publ. No. 69C-30-C, 60-66.

11 Burke, P.J. The output of a queueing system. Oper. Res., 4, 6 (Dec. 1956), 699-704.

12 Buzen, J. Analysis of system bottlenecks using a queueing network model. ACM-SIGOPS Workshop on System Performance Evaluation, ACM, N.Y., Apr. 1971, 82-103.

13 Chang, W. A queuing model for a simple case of time sharing. IBM Sys. J., 5, 2 (1966), 115-125.

14 Chang, W. Queues with feedback for time-sharing computer system analysis. Oper. Res., 16, 3 (June 1968), 613-627.

15 Chang, W. Single server queuing processes in computing systems. IBM Sys. J., 9, 1 (Jan. 1970), 36-71.

16 Chang W. and Wong, D.J. Analysis of real time multiprogramming. JACM, 12, 4 (Oct. 1965), 581-588.

17 Coffman E.G. Stochastic Models of Multiple and Time-Shared Computer Operations. Ph.D. Thesis, Dept. of Engineering, Univ. of Calif., Los Angeles, Cal., June 1966. (Available from National Technical Information Service, Springfield, Va., as AD 636 976.)

18 Coffman, E.G. Studying multiprogramming through the use of queueing theory. Datamation, 13, 6 (June 1967), 47-54.

19 Coffman, E.G. An analysis of computer operations under running time priority disciplines. In Klerer, M. and Reinfelds, J. (ed). Interactive Systems for Experimental Applied Mathematics, Academic Press, N.Y., 1968, 257-270.

20 Coffman, E.G. Analysis of two time-sharing algorithms designed for limited swapping. JACM, 15, 3 (July 1968), 341-353.

21 Coffman, E.G. Analysis of a drum input/output queue under scheduled operation in a paged computer system. JACM, 16, 1 (Jan. 1969), 73-90.

22 Coffman, E.G. and Kleinrock, L. Feedback queueing models for time-shared systems. JACM, 15, 4 (Oct. 1968), 549-576.

23 Coffman, E.G. and Krishnamoorthi, B. Preliminary Analysis of Time-Shared Computer Operation. Doc. SP-1719, Sys. Dev. Corp., Santa Monica, Cal., 1964.

24 Coffman, E.G. and Muntz, R.R. Models of pure time-sharing disciplines for resource allocation. Proc. ACM 1969 Nat. Conf., ACM Publ. No. P-69, 217-228.

25 Coffman, E.G., Muntz, R.R., and Trotter, H. Waiting time distributions for processor sharing systems. JACM, 17, 1 (Jan. 1970), 123-130.

26 Cox, D.R. and Smith, W.L. Queues. Methuen and Co., London, 1961.

27 Delbrouck, L.E.N. A feedback queueing system with batch arrivals, bulk service, and queue-dependent service time. JACM, 17, 2 (Apr. 1970), 314-323.

28 Denning, P.J. Effects of scheduling on file memory operations. Proc. AFIPS 1967 SJCC, Vol. 30, Thompson Books, Wash., D.C., 9-21.

29 Denning, P.J. The working set model for program behavior. CACM, 11, 5 (May 1968), 323-333.

30 Denning, P.J. Thrashing: its causes and prevention. Proc. AFIPS 1968 FJCC, Vol. 33, Thompson Books, Wash., D.C., 915-922.

31 Estrin, G. and Kleinrock, L. Measures, models and measurements for time-shared computer utilities. Proc. ACM 1967 Nat. Conf., Thompson Books, Wash., D.C., 85-96.

32 Feller, W. An Introduction to Probability Theory and its Applications, Vol. 1, Third ed., Wiley and Sons, N.Y., 1968.

33 Fenichel, R.R. and Grossman, A.J. An analytic model of multiprogrammed computing. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., 717-721.

34 Fife, D.W. An optimization model for time-sharing. Proc. AFIPS 1966 SJCC, Vol. 30, Thompson Books, Wash., D.C., 97-104.

35 Fife, D.W. and Rosenberg, R.S. Queueing in a memory-shared computer. Proc. ACM 1964 Nat. Conf., Thompson Books, Wash., D.C., H1-1 - H1-13.

36 Fife, D.W. and Smith, J.L. Transmission capacity of disk storage systems with concurrent arm positioning. IEEE Trans. on Elect. Comp., EC-14, 4 (Aug. 1965), 575-582.

37 Finch, P.D. Cyclic queues with feedback. J. Roy. Stat. Soc., Ser. B, 21, 1 (1959), 153-157.

38 Finch, P.D. The output process of the queueing system M/G/1. J. Roy. Stat. Soc., Ser. B, 21, 2 (1959), 375-380.

39 Frank, H. Analysis and optimization of disk storage devices for time-sharing. JACM, 16, 4 (Oct. 1969), 602-620.

40 Gaver, D.P. Probability models for multiprogramming computer systems. JACM, 14, 3 (July 1967), 423-438.

41 Gordon, W.J. and Newell, G.F. Closed queuing systems with exponential servers. Oper. Res., 15, 2 (Apr. 1967), 254-265.

42 Gordon, W.J. and Newell, G.F. Cyclic queuing systems with restricted queue lengths. Oper. Res., 15, 2 (Apr. 1967), 266-277.

43 Gordon, W.J. and Newell, G.F. Acknowledgment. Oper. Res., 16, 6 (Dec. 1967), 1182.

44 Greenberger, M. The priority problem and computer time-sharing. Management Science, 12, 11 (July 1966), 888-906

45 Harrison, G. Message buffering in a computer switching center. IEEE Trans. on Communication and Electronics, 82, (Sept. 1963), 532-534.

46 Hunt, G.C. Sequential arrays of waiting lines. Oper. Res., 4, 6 (Dec. 1956), 674-683.

47 Jackson, J.R. Networks of waiting lines. Oper. Res., 5, 4 (Aug. 1957), 518-521.

48 Jackson, J.R. Jobshop-like queueing systems. Management Science, 10, 1 (Oct. 1963), 131-142.

49 Jackson, R.R.P. Queueing processes with phase-type service. J. Roy. Stat. Soc., Ser. B, 18, 1 (1956), 129-132.

50 Kendall, D.G. Stochastic processes occuring in the theory of queues and their analysis by the method of the imbedded Markov chain. Ann. Math. Stat., 24 (1953), 338-354.

51 Kleinrock, L. Analysis of a time-shared processor. Naval Res. Logistics Quart., 11, 1 (Mar. 1964), 59-73.

52 Kleinrock, L. Sequential processing machines (S.P.M.) analyzed with a queuing theory model. JACM, 13, 2 (Apr. 1966), 179-193.

53 Kleinrock, L. Time-shared systems: a theoretical treatment. JACM, 14, 2 (Apr. 1967), 242-261.

54 Kleinrock, L. Certain analytic results for time-sharing processors. Proc. IFIP 1968 Cong., Vol. 2, North-Holland Publ. Co., Amsterdam, 838-845.

55 Kleinrock, L. On swap time in time-shared systems. Digest IEEE 1969 Comp. Group Conf., IEEE Publ. No. 69C-30-C, 37-41.

56 Kleinrock, L. A continuum of time-sharing scheduling algorithms. Proc. AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N.J., 453-458.

57 Kleinrock, L. and Coffman, E.G. Distribution of attained service times in time-shared systems. J. Comp. and Sys. Sciences, 1, 3 (Oct. 1967), 287-298.

58 Koenigsberg, E. Cyclic queues. Operational Res. Quart., 9, 1 ((1958), 22-35.

59 Krishnamoorthi, B. The stationary behavior of a time-sharing system under Poisson assumptions. OPSEARCH-J. Oper. Res. Soc. of India, 3, 3 (1966), 101-117.

60 Krishnamoorthi, B. and Wood, R.C. Time-shared computer operations with both interarrival and service times exponential. JACM, 13, 3 (July 1966), 317-338.

61 McKinney, J.M. A survey of analytic time-sharing models. Computing Surveys, 1, 2 (June 1969), 105-116.

62 Moore, C.G. Network Models for Large-Scale Time-Sharing Systems. Ph.D. Thesis, Univ. of Mich., Ann Arbor, Mich., April 1971.

63 O'Brien, G.G. The solution of some queuing problems. J. Soc. Indust. Applied Math., 2, 3 (Sept. 1954), 133-142.

64 Patel, N.R. A Mathematical Analysis of Computer Time-Sharing Systems. Masters Thesis, Dept. of E.E., Mass. Inst. of Tech., Cambridge, Mass., June 1964.

65 Phipps, T.E., Jr. Machine repair as a priority waiting-line problem. Oper. Res., 4, 1 (Feb. 1956), 76-85.

66 Rasch, P.J. A queueing theory study of round-robin scheduling of time-shared computer systems. JACM, 17, 1 (Jan. 1970), 131-145.

67 Reich, E. Waiting times when queues are in tandem. Ann. Math. Stat., 28, 3 (1957), 768-773.

68 Sakata, M., Noguchi, S. and Oizumi, J. Analysis of a processor shared queueing model for timesharing systems. Proc. of Second Hawaii Int. Conf. on Sys. Science, Univ. of Hawaii, Honolulu, 1969, 625-628.

69 Scherr, A.L. An Analysis of Time-Shared Computer Systems. Ph.D. Thesis, Dept. of E.E., Mass. Inst. of Tech., Cambridge, Mass., June 1965. (Available from the MIT Press, Cambridge, Mass.)

70 Schrage, L.E. The queue M/G/1 with feedback to lower priority queues. Management Science, 13,7 (Mar. 1967), 466-474.

71 Schrage, L.E. Analysis and optimization of a queueing model of a real time computer control system. IEEE Trans. on Comp., C-18, 11 (Nov. 1969), 997-1003.

72 Schrage, L.E. and Miller, L.W. The queue M/G/1 with the shortest remaining processing time discipline. Oper. Res., 14, 4 (Aug. 1966), 670-684.

73 Seaman, P.H., Lind, R.A. and Wilson, T.L. An analysis of auxiliary-storage activity. IBM Sys. J., 5, 3 (1966), 158-170.

74 Sharma, R.L. Analysis of a scheme for information organization and retrieval from a disk file. Proc. IFIP 1968 Cong., Vol. 2, North-Holland Publ. Co., Amsterdam, 853-859.

75 Shemer, J.E. Some mathematical considerations of time-sharing scheduling algorithms. JACM, 14, 4 (Apr. 1967), 262-272.

76 Smith, J.L. An analysis of time-sharing computer systems using Markov models. Proc. AFIPS 1966 SJCC, Vol. 28, Thompson Books, Wash., D.C., 87-95.

77 Smith, J.L. Multiprogramming under a page on demand strategy. CACM, 10, 10 (Oct. 1967), 636-646.

78 Takacs, L. Introduction to the Theory of Queues. Oxford Univ. Press, N.Y., 1962.

79 Tanaka, H. An analysis of on-line system using parallel cyclic queues. Denshi Tsushin Gakkai Rombunshi (J. of the Assoc. of Electronics and Communication of Japan),

53-C, 10 (Oct. 1970), 756-764, (Japanese).

80 Wallace, V.L. and Mason, D.L. Degree of multiprogramming in page-on-demand systems. CACM, 12, 6 (June 1969), 305-318.

81 Wallace, V.L. and Rosenberg, R.S. Markovian models and numerical analysis of computer system behavior. Proc. AFIPS 1966 SJCC, Vol. 28, Thompson Books, Wash., D.C., 141-148.

82 Walter, E.S. and Wallace, V.L. Further analysis of a computing center environment. CACM, 10, 5 (May 1967), 266-272.

83 Weingarten, A. Storage requirements for a message switching computer. IEEE Trans. on Communications Sys., CS-12, 2 (June 1964), 191-195.

84 Weingarten, A. The Eschenbach drum scheme. CACM, 9, 7 (July 1966), 509-512.

85 Weingarten, A. The analytic design of real-time disk systems. Proc. IFIP 1968 Cong., Vol. 2, North-Holland Publ. Co., Amsterdam, 860-866.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

Harvard University
Division of Engineering and Applied Physics
Cambridge, Massachusetts 02138

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

N/A

3. REPORT TITLE

QUEUEING NETWORK MODELS OF MULTIPROGRAMMING

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

None

5. AUTHOR(S) *(First name, middle initial, last name)*

Jeffrey P. Buzen

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
August 1971	248	85

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
F19628-70-C-0217	
b. PROJECT NO.	ESD-TR-71-345
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
THESIS Div. of Eng. and Applied Physics Harvard University	Deputy for Command and Management Systems Hq Electronic Systems Division (AFSC) L G Hanscom Field, Bedford, Mass. 01730

13. ABSTRACT

A model is developed which represents the behavior of multiprogrammed computer systems in terms of a network of interdependent queues. This model, which is known as the central server model of multiprogramming, is first analyzed mathematically and then applied to three problems in operating system design. These are: the optimal choice of buffer size for tape-like devices; the optimal allocation of processing requests among a set of functionally equivalent peripheral processors such as disks and drums; the optimal selection of the degree of multiprogramming in demand paging systems.

A series of computational algorithms are developed to supplement the analytic work. These algorithms can be used to obtain the marginal distributions and expected queue lengths for a large class of queueing network models.

DD FORM 1 NOV 65 1473

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computers						
Multiprogramming						
Operating System Design						
Optimization of Computer Systems						
Performance Evaluation						
Queueing						
Queueing Networks						